TEL AVIV UNIVERSITY אוניברסיטת תל-אביב

The Raymond and Beverly Sackler Faculty of Exact Sciences

The Blavatnik School of Computer Science

On Generic Computational Models

Thesis submitted for the degree of Doctor of Philosophy

by

Evgenia Falkovich

This work was carried out under the supervision of

Professor Nachum Dershowitz

Submitted to the Senate of Tel Aviv University

October 2014

Acknowledgements

I wish to thank my supervisor, Professor Nachum Dershowitz, a lot. First of all for his incredible ability to ask unexpected questions, which might seem irrelevant at first glance, but deep, fundamental and "discovering the whole new world" on the second one. Also, without his endless kindness and patience this thesis would never have seen the light of day.

I wish to thank also Yuri Gurevich, Neil Immerman, Gilles Dowek and David Harel for the useful talks that we had, for their good ideas and for their support and trust in the importance of my research.

Last but not least, thanks go to my family: to my parents for their support and development of my curiosity ever since I was a kid; to my husband for unconditional belief in me, even at my breakdown points; and to my cat, who spent hours on my knees reading books and papers with me, and yelling at me to get back to work, any time that my break was too long.

Abstract

The mechanization of computation has challenged humans for centuries. The abacus is believed to have been invented by the Babylonians in the 24th century B.C.E. Archimedes used the mechanical principle of balance to calculate mathematical problems, such as the number of grains of sand in the universe, in the 3rd century B.C.E. The Antikythera mechanism, which is believed to be one of the first mechanical analog computers, was constructed in the 1st century B.C.E. and has been reconstructed in recent times. It included displays of the phases of the moon, eclipse predictions, and the positions of the visible planets.

In the early 1900s, David Hilbert posed, among other problems, his famous Entscheidungsproblem. This was a research challenge for finding a procedure that for any given logical expression can decide its validity or satisfiability by finitely many operations. On his way to a solution to this problem, Alan Turing in his revolutionary paper, presented a formal computational model, called today "Turing machines". Over the next few decades, many brilliant minds, like Kolmogorov, Schönhage, Cook, and Reckhow suggested alternative models for effective computation.

But since then, the notion of what constitutes computation has evolved (and keeps evolving) rapidly. Today, computation comes in many flavors: the classical discrete, sequential version; analog or hybrid setups; and the more recent parallel and concurrent varieties. Moreover, virtually any natural process may be viewed as an evolving system of hybrid (mixed discrete and analog) computation. Modern evolving systems—be they physical, biological, or computational—are typically viewable on many distinct levels of abstraction.

Our overarching goal in this study is to analyze the fundamental aspects of these different computational paradigms, unite them under a common formalism, and understand their implications for questions of effectivity and complexity. In particular, we study parallel, concurrent, and hybrid computational mechanisms.

It has been convincingly argued by Gurevich (presaged by Post) that logical structures are the right way to view evolving algorithmic states, just as they are ideal for

capturing the salient features of static entities. Based on that insight and a formalization of what it means for an algorithm to be governed by a finite text, Gurevich provided the most general and generic definition of what constitutes a classical sequential algorithm. His model captures the algorithmic behavior on whatever level of abstraction is natural for the particular algorithm. In this work, we develop his ideas further. We provide a similar-style definition for generic parallel algorithms and prove that the evolution of any system that satisfies our axiomatization is indeed mechanical.

We extend the definition of effectiveness, given by Boker and Dershowitz, to cover effectiveness relative to oracles (as suggested by Turing). We also use their original definition of effectiveness to provide a general notion of complexity of effective algorithms. With that in hand, we prove two results: any effective classical algorithm may be simulated by a random-access machine with only constant factor time overhead, and that effective parallel algorithms can be simulated by a parallel random-access machine with only polylog overhead in time and number of processors. The latter also leads to the conclusion that Turing poly-space is equivalent to parallel poly-time, given that algorithms may not have more than an exponential number of agents.

In the same groundbreaking paper, Turing defined a universal machine, which may simulate any other computation. In this work we suggest a similar notion of universality over arbitrary domains. We also address the issue of "honesty" of representations. Another issue of honesty that we address is in the representation of effective domains. Since effectiveness presupposes order; effective domains may "hide" some unexpected information. Also, ordered domains are not always natural; for example, a graph does not naturally have any order on its nodes. As a way to overcome this, Gurevich suggested the definition of general "fair" unordered domains. We, in turn, define an alternative computational model, the dynamic cellular automaton, and prove that those automata simulate unordered domains. Thus, this model may be used for fair unordered computations.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

The notion of what constitutes computation has evolved rapidly in recent decades. Today, computation comes in many flavors: the classical discrete, sequential version; analog or hybrid setups; and the more recent parallel and concurrent varieties. Our overarching goal in this study is to analyze the fundamental aspects of these different computational paradigms and to understand their implications for questions of effectivity and complexity. This foundational line of investigation has the potential to impact the design of specification, prototyping, and programming languages and tools for these classes of computation.

1.1 Algorithmic Computation

Donald Knuth writes [78]:

> Algorithms are concepts which have existence apart from any programming language.... Algorithms were present long before Turing et al. formulated them, just as the concept of the number "two" was in existence long before the writers of first grade textbooks and other mathematical logicians gave it a certain precise definition.

Indeed, algorithms existed long before Turing invented his machines. Though we lack detailed information about the mathematics developed by ancient Egyptians beginning some 5000 years before the present, we do know that they used a decimal representation for natural numbers and that they had a linear algorithm for multiplication based on the ability to multiply and divide by two along with addition. Much

more is known about the mathematics developed by ancient Babylonians some 4000 years ago. As may be seen from clay tablets found in the 19th century, they used a sexagesimal (base 60) numeral system. In [77], Knuth describes some of their algorithms. They did not have algebraic notations, but they described formulas as step-by-step lists of rules for evaluation. This might be considered as a form of ancient programming. Those computations are linear. As Knuth states, there is only one slight intimation of the use of conditions in computation. That is in the division algorithm, were computation proceeds differently if the reciprocal of the divisor does not appear in the table. He suggests that modern mathematics is deeply rooted in such ancient knowledge.

Conditionals are, however, found widely in the mathematics of the ancient Greeks, known from about the third century B.C.E. A prime example is the well-known Euclidian algorithm for greatest common divisor. Euclidian geometry uses algorithms in terms of ruler and compass operations, which are perfect examples of non-effective algorithms, at a time when the notion of algorithm was not yet well-defined.[1]

In [62], Gurevich suggested a very general axiomatization of sequential computation, which corresponds to the classical notion of algorithm. This formalization works over arbitrary domains. Gurevich proved that any transition system satisfying these axioms can be described syntactically as an *abstract state machine* (ASM), a generic model of computation, over the same signature. Roughly speaking, this machine has two main operations: it can compare the values of terms—to verify the current state, and it can process assignments—to update the current state.

1.2 Mechanical Computation

Many mechanical devices have been devised for specific computational purposes, ranging from the abacus to Gottfried Wilhelm von Leibniz's Step Reckoner for multiplication to various arithmetical calculators of the twentieth century, such as the wonderful Curta [111].

Perhaps the most remarkable ancient device is the recently re-constructed Antikythera mechanism, which included displays of the phases of the moon, eclipse predictions, and the positions of the visible planets [110].

The first suggestion of a *universal* computing device appears to be Charles Bab-

[1] Al-Khwarizmi's *Algoritmi de numero Indorum* was published in the 9th century.

bage's Analytical Engine, about which Ada Augusta Lovelace presciently made the following remarks [83]:

> [The Engine is for] developing and tabulating any function whatever.... The engine [is] the material expression of any indefinite function of any degree of generality and complexity.
>
> Many persons who are not conversant with mathematical studies imagine that because the business of the engine is to give its results in numerical notation, the nature of its processes must consequently be arithmetical and numerical rather than algebraical and analytical. This is an error. The engine can arrange and combine its numerical quantities exactly as if they were letters or any other general symbols; and in fact it might bring out its results in algebraical notation were provisions made accordingly.

One more recent design: In 1982, Fredkin and Toffoli [48] described a mechanical model built of ideal billiard balls and polygonal obstacles and proved that it can, conceptually at least, simulate any polynomial-space *reversible* Turing machine.

1.3 Analog Computation

Noted physicist, Freeman Dyson, writes in *Edge* (March 13, 2001):

> The two ways of processing information are analog and digital. An LP record gives us music in analog form, a CD gives us music in digital form. A slide-rule does multiplication and division in analog form, an electronic calculator or computer does them in digital form.

Dyson's examples illustrate two dimensions of what is called analog. A slide-rule is analog, since computations work with lengths of intervals that are in analogy with the real values of interest. Thus, it computes approximations involving continuous quantities, but the steps in a computation are discrete and sequential. This aspect of analog computation is called continuous space, since it operates over an uncountable domain. A compact disc records and produces discrete values, but it is continuously spinning and so its operations occur at moments of continuous time. A vinyl record, on the other hand, not only operates in real, continuous time, but produces real-valued displacements at each moment of time, in such a way that the motions of the

needle are analogous to sound waves. And, of course, an ordinary calculator is purely digital. Hybrid systems, such as robots, combine aspects of both the digital and analog computing paradigms.

Probably the best known "universal" continuous-time machine is Vannevar Bush's landmark 1931 Differential Analyzer [22]. Preceding that, there were special purpose analog devices, including Blaise Pascal's 1642 Pascaline and Johann Martin Hermann's 1814 Planimeter. In 1941, Shannon [101] proposed the *General Purpose Analog Computer (GPAC)* as a theoretical model of the differential analyzer and asserted that it generates precisely the class of differentially algebraic functions [88]. A more robust class of GPACs has been defined recently by Graça and Costa [59]. Bournez [20] has shown that, by considering a notion of computation inspired by recursive analysis [58], the GPAC-computable functions are precisely the computable functions over reals. In particular, the Gamma function is computable in this sense, but not in Shannon's original model [96]. Rubel [97] proposed an extension of Shannon's original GPAC, called the *Extended Analog Computer*, with additional operations to solve boundary value problems or to take certain infinite limits, and this extension has been implemented [84]. For surveys of analog computing, see [19, 81].

Continuous-space and continuous-time computing are both gaining in importance, especially in this modern world of complex embedded systems. Various models of these schemes have been proposed. For example, the Blum-Shub-Smale model [11] is a sequential programming language working in continuous space and allowing equality tests between real values. Real-time programming (e.g. [104]) is a relatively mild form of continuous-time programming, in which it is necessary to take into account the time required for each operation.[2] Continuous-time computational models also include neural networks [60], and systems that can be built using electronic analog devices.

1.4 Natural Computation

Evolving systems—be they physical, biological, or computational—are typically viewable on many distinct levels of abstraction. Let us imagine some closed ecosystem as an example. An *ecologist* views species, populations, and their interactions; population growth and shrinkage may be modeled, say, by predator-prey and other resource equa-

[2] We are using the term "real-time programming" for "time aware" programming, in general, not for "permanently ready" compilers.

tions. A *biologist* takes a different viewpoint, based on the individual organisms; she may develop a kinetic model for swarming behavior, for instance. On a lower level still, a *biochemist* sees interacting cell systems; he might use a diffusion-reaction equation to describe the development of the colorings on an animal's coat. The *chemist* looks at reactions on the molecular level; the *physicist* sees atoms and their constituents. The common denominator of all these views is one of a complex of objects that evolve over time and that interact with each other and with their environment according to a set of rules. As Galileo observed in *Il Saggiatore*, the "manual" of the universe is written in mathematical language. It is this generic notion of a *system of interacting objects* that we seek to capture in this research.

1.5 Effective Computation

In 1900, David Hilbert posed, among other problems, the research challenge of how to effectively determine whether any given polynomial with rational coefficients has rational roots [70]. Later, he and Wilhelm Ackermann underscored the importance of the decision problem for validity of formulæ in (first-order predicate) logic, which they called the *Entscheidungsproblem* [71, pp. 73–74].

Hilbert was seeking an effective procedure that could solve every instance of the validity question, positively or negatively: "We assume that we have the capacity to name things by signs, that we can recognize them again. With these signs we can then carry out operations that are analogous to those of arithmetic and that obey analogous laws" (quoted in [107]). This stressed the following question: "What is an effective algorithm?".

In 1936, Alonzo Church suggested that the recursive functions, or the computationally equivalent lambda-definable numeric functions, capture the intended concept of "effectively calculable" procedure [26, p. 356]. Alan Turing [113] suggested that all computable functions may be computed by a formal computational machines, that he suggested, and that we call today "Turing machines", a model that has gained almost universal acceptance as synonymous with effective computation. To quote Turing in 1948 [115]:

> Logical Computing Machines [i.e. what are today called "Turing machines"] can do anything that could be described as "rule of thumb" or "purely me-

> chanical". This is sufficiently well established that it is now agreed amongst logicians that "calculable by means of an LCM" is the correct rendering of such phrases.

Turing machines provide a computational model for strings and recursive functions for the natural numbers. But computability is a more general notion than recursiveness or Turing computability. Boker and Dershowitz [14] proposed a principled way to compare the computational power of different models of computation. This led to an ASM-style axiomatization of *effective* algorithms [15, 41]. The main difference of this class is that the domain of each machine is constructible from its signature. Unlike Turing's analysis [113], and subsequent generalizations [52, 79, 80, 105, 106, 108, 109], those axioms of effective computation are, at the same time, both formal and generic. They are formal in that they may be cast as precise mathematical statements; they are generic in that they apply to computations with arbitrary states and arbitrary programmable transitions.

Beyond that, Turing extended the notion of computability to devices provided with oracles that "magically" provide answers to questions for which there may be no effective means of providing answers. We extend the definition given by Boker and Dershowitz to encompass effectiveness relative to oracles (Chapter 3).

1.6 Parallel Computation

In 1978, Fortune and Wyllie endowed random-access machines with multiple processors working simultaneously governed by a common clock [47]. We call this model today the *parallel random–access machine (PRAM)*. This model has gained much attention and different parallel models have been developed. In 1979 [68], Hemmerling described a model with several finite-state automata working on one common tape. In 1984 [120], Wiedermann generalized this idea to parallel Turing machine.

Another effort was made to generalize a notion of parallel computation. Goldschlager in [56] described a model with almost identical processors that all start simultaneously at the start of computation and communicate via shared memory. Galil and Paul [51] described a model of computation that has "similar" processors that start up at runtime and communicate via direct processor-to-processor links. From an intuitive point of view, parallel computation consists of a collection of processors that all execute

the same algorithm and in some way co'operate to exchange results.

All the above models work over classical domains: strings or natural numbers. In [8], Blass and Gurevich [6, 8] successfully characterized parallel algorithms within the abstract-state-machine framework. Their approach, being formal and generic, is not easy to restrict to the effective case, due to a strong usage of multisets. So we propose an alternative characterization of parallel algorithms, also within the abstract-state-machine framework. Our model is simpler than that of Blass and Gurevich for the cases we consider, but is restricted to shared memory co'operation only and also restricts the number of child processes. On the other hand, in our model, algorithms need not deal at all with process identifiers and the number of processes at the initial step is unrestricted and may be of any cardinality. We provide an alternative programming language, slightly different from what Blass and Gurevich suggested, and prove that evolution of any computational system that satisfies our axioms may be described by a finite rule in our language (Chapter 6).

1.7 Universal Computation

In the same groundbreaking paper [113] in which he invented the Turing machine and settled the Entscheidungsproblem, Turing suggested a notion of universal machine—one machine that can do any computation. Again, the notion of universal machine was considered over two classical domains: Davis [31] and Rogers [94] have proposed general definitions of universality for Turing machines and for partial recursive functions, respectively. We extend the notion of universality to arbitrary sets of functions over arbitrary domains, while addressing oft-ignored questions of "honesty" of representations of domains (Chapter 4).

1.8 Complexity of Computation

The inefficiency of Turing machines led to the development of more efficient models. In 1971, Hartmanis [67] and Cook and Reckhow [29] developed the random-access register machine (RAM) model for the purpose of measuring computational complexity of computer algorithms. This theoretical model is close in spirit to the design of modern, von Neumann architecture, computers and serves as a more realistic measure of (asymptotic) time and space resource usage than do Turing's machines. Indeed, the

RAM model has become the standard machine model for the analysis of concrete algorithms. It remains conceivable, however, that there exists some sort of model that is more sophisticated than RAMs, one that allows for even more time-wise efficient algorithms, yet ought still be considered "reasonable". We prove that this is not the case (Chapter 7).

Any effective model by its nature works over an ordered domain. Being a non-trivial relation, order may hide some non-trivial and even non-computable information. And an algorithm may take advantage on that order. So our next challenge is to prove the following statement: *any function computable over an unordered domain can be computed by a RAM with only a small time overhead.* To do so, we will use algorithms over an unordered domain, as defined in [9].

A further challenge is to exploit the computational power of effective non-classical models like parallel and distributed machines. As explained by Parberry [85]: *The Extended Church-Turing Thesis states . . . that time on all "reasonable" machine models is related by a polynomial.* To establish this version for parallel machines, we use our formalization of parallel algorithms, but restrict it to a class of effective parallel algorithm. We investigate if those algorithms are equivalent to classical parallel models, like the PRAM.

1.9 Outline of this Thesis

The remainder of this dissertation is organized as follows.

Chapter 2 *What is a Sequential Algorithm?*

Evolving algebras, invented by Yuri Gurevich [61], constitute a most general model of computation, one that can operate on any desired level of abstraction of data structures and native operations. All (ordinary) models of computation are instances of this one generic paradigm. Later, Gurevich [62] axiomatized sequential algorithms and proved that the evolution of the states of any such algorithm may be captured by abstract-state-machine (ASM) programs, proving that evolution may be described in a purely mechanical way. In this chapter, we give an overview of the foundational considerations underlying this model, on top of which our work proceeds (based on the exposition in [5, 16, 41]).

Chapter 3 *What is an Effective Algorithm?*

In [13, 16], Boker and Dershowitz presented a constructor-based axiomatization of effectiveness, which is both formal and generic, based on Gurevich's axiomatization of algorithms, in general. We will adopt that formalization in this work. In this chapter, we generalize that notion with a generic definition of effectiveness relative to oracle operations. This work was published in [37].

Chapter 4 *Universality.*

Alan Turing, in his groundbreaking 1936 paper on the undecidability of the Halting Problem and the Entscheidungsproblem [113], also invented the notion of a universal machine. There are two domains that are standard in discussions of computability: (1) finite strings over finite alphabets—for which the Turing-computable partial functions are the effective ones, and (2) the natural numbers—for which the effective functions are identified with the partial recursive ones. We generalize the notion of universality to arbitrary sets of functions over arbitrary domains, while addressing oft-ignored questions of "honesty" of representations of domains. This work was described in [38].

Chapter 5 *Generic Evolving Systems.*

In [34], Dershowitz presented a scheme of generic evolving systems. Those systems are composed of coöperating cells. Each cell has some local data and may evolve in its

own way. Systems can have shared memory, which is accessible by all cells. In addition, cells may communicate via message passing. Some cells are connected by channels and may use those connections to send each other messages. In this section, we present an extended version of this model (a publication is being prepared).

Chapter 6 *What is a Parallel Algorithm?*

In this chapter, we represent parallel algorithms as a special case of generic evolving systems, where at each tick of a common clock every cell executes exactly one step of a common sequential algorithm and may also create a bounded number of child cells. We prove, similar to [62], that evolution of such a system may be captured by a finite program, which we call a *parallel ASM program*. This work was published in [40].

Chapter 7 *Extended Computational Thesis.*

In Chapter 2, we describe a notion of classical generic algorithm and restrict it to the effective class in Chapter 3. In the first part of this chapter, we define a generic notion of complexity for classical effective algorithms and prove that any such algorithm can be simulated by a RAM with only a constant factor overhead in time. A somewhat weaker result was published in [36], a journal version of the improved result has been submitted to [35]

In Chapter 6, we provide a notion of a generic parallel algorithm and its restriction to a class of effective ones. In the second part of this chapter, we define an appropriate notion of complexity for *parallel* effective algorithms. We prove that any parallel effective algorithm with $P(n)$ cells and time complexity $T(n)$ may be simulated by an arithmetic parallel-RAM (PRAM) with time overhead polylog $T(n)$ polylog $P(n)$. We then deduce that any parallel effective algorithm with no more than an exponential number of cells may be simulated by a PRAM with only polynomial-time cost. From that we conclude that polynomial parallel time is equivalent to polynomial Turing space, as been conjectured (these results will be sent to a journal).

Chapter 8 *Cellular Automata.*

In this chapter, we present a special case of generic evolving systems, called dynamic cellular automata. In this model, each cell is in one of a predefined finite set of states—which we call *colored* cells. At each step, one active cell makes a bounded number of changes in its closed neighborhood. We show that this model captures classical

unordered algorithms. This work was published in [39].

Chapter 9 *Continuous Time.*

We believe that continuous-time processes can be formalized within a framework of evolving logical structures. The significance of our efforts lies in the building of foundations for the design of specification and programming languages and tools for analog and hybrid computing, paradigms that are crucial in today's world. Capturing the notion of algorithm and computation for analog systems is also a first step towards a better understanding of computability theory for continuous-time systems.

In this chapter, we adapt and extend ideas from work on ASMs to the analog case, that is to say, from notions of algorithms for digital models to analogous notions for analog systems. We provide a partial axiomatization for generic analog models. This work was published in [21].

Chapter 10 *Conclusions and Future Work.*

We conclude with a brief summary of the contributions of this research and some ideas for future work.

Chapter 2

What is a Sequential Algorithm?

2.1 Background

Abstract state machines (ASMs), invented by Yuri Gurevich [61], constitute a most general model of computation, one that can operate on any desired level of abstraction of data structures and native operations. All ordinary models of computation are instances of this one generic paradigm. Here, we give an overview of the foundational considerations underlying the model (borrowed from [33][1]).[2]

Programs (of the sequential, non-interactive variety) in this formalism are built from three components:

- There are generalized assignments $f(s_1, \ldots, s_n) := t$, where f is any function symbol (in the vocabulary of the program) and the s_i and t are arbitrary terms (in that vocabulary).
- Statements may be prefaced by a conditional test, **if** C **then do else do** , where C is a propositional combination of equalities between terms.
- Program statements may be composed in parallel, following the keyword **do**, short for **do in parallel**.

An ASM program describes a single transition step; its statements are executed repeatedly, as a unit, until no assignments have their conditions enabled. (Additional constructs beyond these are needed for interaction and large-scale parallelism, which are not addressed in this chapter).

[1] With permission.
[2] For a video lecture of Gurevich's on this subject, see http://www.youtube.com/v/7XfA5EhH7Bc.

Algorithm 1 An abstract-state-machine program for sorting.

$$\textbf{if } j = n \textbf{ then if } \ i+1 \neq n \textbf{ then do } \begin{cases} i := i+1 \\ j := i+2 \end{cases}$$
$$\textbf{else do } \begin{cases} \textbf{if } F(i) > F(j) \textbf{ then do } \begin{cases} F(i) := F(j) \\ F(j) := F(i) \end{cases} \\ j := j+1 \end{cases}$$

Algorithm 2 An abstract-state-machine program for bisection search.

$$\textbf{if } |b-a| > \varepsilon \textbf{ then do } \begin{cases} \textbf{if } \operatorname{sgn} f((a+b)/2) = \operatorname{sgn} f(a) \textbf{ then } a := (a+b)/2 \\ \textbf{if } \operatorname{sgn} f((a+b)/2) = \operatorname{sgn} f(b) \textbf{ then } b := (a+b)/2 \end{cases}$$

As a simple example, consider the program shown as Algorithm 1, describing a version of selection sort, where $F(0), \ldots, F(n-1)$ contain values to be sorted, F being a unary function symbol.

Initially, $n \geq 1$ is the quantity of values to be sorted, i is set to 0, and j to 1. The brackets indicate statements that are executed in parallel. The program proceeds by repeatedly modifying the values of i and j, as well as of locations in F, referring to terms $F(i)$ and $F(j)$. When all conditions fail, that is, when $j = n$ and $i + 1 = n$, the values in F have been sorted vis-à-vis the black-box relation "$>$". The program halts, as there is nothing left to do. (Declarations and initializations for program constants and variables are not shown.)

This sorting program is not partial to any particular representation of the natural numbers 1, 2, etc., which are being used to index F. Whether an implementation uses natural language, or decimal numbers, or binary strings is immaterial, as long as addition behaves as expected (and equality and disequality, too). Furthermore, the program will work regardless of the domain from which the values of F are drawn (be they integers, reals, strings, or what not), so long as means are provided for evaluating the inequality ($>$) relation.

Another simple ASM program is shown in Algorithm 2. This is a standard bisection search for the root of a function, as described in [57, Algorithm #4]. The point is that this abstract formulation is, as the author of [57] wrote, "applicable to any continuous function" over the reals—including ones that cannot be programmed.

What is remarkable about ASMs is that this very simple model of computation suffices to precisely capture the behavior of the whole class of ordinary algorithms

over any domain. The reason is that, by virtue of the abstract state machine (ASM) representation theorem of [62] (Theorem 2 below), any algorithm that satisfies three very natural "Sequential Postulates" can be *step-by-step, state-for-state* emulated by an ASM. Those postulates, articulated in Section 2.2, formalize the following intuitions: (I) an algorithm is a state-transition system; (II) given the algorithm, state information determines future transitions and can be captured by a logical structure; and (III) state transitions are governed by the values of a finite and input-independent set of terms.

The significance of the Sequential Postulates lies in their comprehensiveness. They formalize which features exactly characterize a classical algorithm in its most abstract and generic manifestation. Programs of all models of effective, sequential computation satisfy the postulates, as do idealized algorithms for computing with real numbers (e.g. Algorithm 2), or for geometric constructions with compass and straightedge (see [91] for examples of the latter).

Abstract state machines are a computational model that is not wedded to any particular data representation, in the way, say, that Turing machines manipulate strings using a small set of tape operations. The Representation Theorem, restated in Section 2.3, establishes that ASMs can express and precisely emulate any and all algorithms satisfying the premises captured by the postulates. For any such algorithm, there is an ASM program that describes precisely the same state-transition function, state after state, as does the algorithm. In this sense, ASMs subsume all other computational models.

It may be informative to note the similarity between the form of an ASM, namely, a single repeated loop of a set of generalized assignments nested within conditionals with the "folk theorem" to the effect that any flowchart program can be converted to a single loop composed of conditionals, sequencing, and assignments, with the aid of some auxiliary variables (see [66]). Parallel composition gives ASMs the ability to perform multiple actions sans extra variables, and to capture all that transpires in a single step of any algorithm.

This versatility of ASMs is what makes them so ideal for both specification and prototyping. Indeed, ASMs have been used to model all manner of programming applications, systems, and languages, each on the precise intended level of abstraction. See [17] and the ASM website (`http://www.eecs.umich.edu/gasm`) for numerous exemplars. ASMs provide a complete means of describing algorithms, whether or not

they can be implemented effectively. On account of their abstractness, one can express generic algorithms, like our bisection search for arbitrary continuous real-valued functions, or like Gaussian elimination, even when the field over which it is applied is left unspecified. AsmL [63], an executable specification language based on the ASM framework, has been used in industry, in particular for the behavioral specification of interfaces (see, for example, [3]).

Church's Thesis asserts that the recursive functions are the only numeric functions that can be effectively computed. Similarly, Turing's Thesis stakes the claim that any function on strings that can be mechanically computed can be computed, in particular, by a Turing machine. More generally, one additional natural hypothesis regarding the describability of initial states of algorithms, as explained in Chapter 3, characterizes the effectiveness of any model of computation, operating over any (countable) data domain (Theorem 32).

On account of the ability of ASMs to precisely capture single steps of any algorithm, one can infer absolute bounds on the complexity of algorithms under arbitrary effective models of computation, as will be seen (Theorem 32) at the end of Section 7.4.

2.2 Sequential Algorithms

The Sequential Postulates of [62] regarding algorithmic behavior are based on the following key observations:

- A state should contain *all* the relevant information, apart from the algorithm itself, needed to determine the next steps. For example, the "instantaneous description" of a Turing machine computation is just what is needed to pick up a machine's computation from where it has been left off; see [113]. Similarly, the "continuation" of a Lisp program contains all the state information needed to resume its computation. First-order structures suffice to model all salient features of states. Compare [87, pp. 420–429].

- The values of programming variables, in and of themselves, are meaningless to an algorithm, which is implementation independent. Rather, it is relationships between values that matter to the algorithm. It follows that an algorithm should work equally well in isomorphic worlds. Compare [52, p. 128]. An algorithm

can—indeed, can only—determine relations between values stored in a state via terms in its vocabulary and equalities (and disequalities) between their values.

- Algorithms are expressed by means of finite texts, making reference to only finitely many terms and relations among them. See, for example, [76, p. 493].

The three postulates given below (from [62], modified slightly as in [5, 7]) assert that a classical algorithm is a state-transition system operating over first-order structures in a way that is invariant under isomorphisms. An algorithm is a prescription for updating states, that is, for changing some of the interpretations given to symbols by states. The essential idea is that there is a fixed finite set of terms that refer (possibly indirectly) to locations within a state and which suffice to determine how the state changes during any transition.

2.2.1 Sequential Time

To begin with, algorithms are deterministic state-transition systems.

Postulate I (Sequential Time)**.** *An algorithm determines the following:*

- *A nonempty set*[3] $\mathcal{S}$ *of* states *and a nonempty subset* $\mathcal{S}_0 \subseteq \mathcal{S}$ *of* initial *states.*
- *A partial* next-state *transition function* $\tau : \mathcal{S} \rightharpoonup \mathcal{S}$*.*

Terminal states $\mathcal{S}_\ddagger \subseteq \mathcal{S}$ are those states X for which no transition $\tau(X)$ is defined.

Having the transition depend only on the state means that states must store all the information needed to determine subsequent behavior. Prior history is unavailable to the algorithm unless stored in the current state.

State-transitions are deterministic. Classical algorithms in fact never leave room for choices, nor do they involve any sort of interaction with the environment to determine the next step. To incorporate nondeterministic choice, probabilistic choice, or interaction with the environment, we will need to modify the above notion of transition.

This postulate is meant to exclude formalisms, such as [55, 89], in which the result of a computation—or the continuation of a computation—may depend on (the limit of) an infinite sequence of preceding (finite or infinitesimal) steps. Likewise, processes in which states evolve continuously (as in analog processes, like the position of a bouncing ball), rather than discretely, are eschewed (see Chapter 9).

[3] Or class; the distinction is irrelevant for our purposes.

Though it may appear at first glance that a recursive function does not fit under the rubric of a state-transition system, in fact the definition of a traditional recursive function comes together with a computation rule for evaluating it. As Rogers [95, p. 7] writes, "We obtain the computation uniquely by working from the inside out and from left to right".

2.2.2 Abstract State

Algorithm states are comprehensive: they incorporate all the relevant data (including any "program counter") that, when coupled with the program, completely determine the future of a computation. States may be regarded as structures with (finitely many) functions, relations, and constants. To simplify matters, relations will be treated as truth-valued functions and constants as nullary functions. So, each state consists of a domain (base set, universe, carrier) and interpretations for its symbols. All relevant information about a state is given explicitly in the state by means of its interpretation of the symbols appearing in the vocabulary of the structure. The specific details of the implementation of the data types used by the algorithm cannot matter. In this sense states are "abstract". This crucial consideration leads to the second postulate.

Postulate II (Abstract State)**.** *The states $\mathcal{S}$ of an algorithm are (first-order) structures over a finite vocabulary $\mathcal{F}$, such that the following hold:*

- *If X is a state of the algorithm, then any structure Y that is isomorphic to X is also a state, and Y is initial or terminal if X is initial or terminal, respectively.*
- *Transitions preserve the domain; that is,* $\mathrm{Dom}\tau(X) = \mathrm{Dom}X$ *for every non-terminal state X.*
- *Transitions respect isomorphisms, so, if $\zeta : X \cong Y$ is an isomorphism of non-terminal states X, Y, then also $\zeta : \tau(X) \cong \tau(Y)$.*

State structures are endowed with Boolean truth values and standard Boolean operations, and vocabularies include symbols for these. As a structure, a state interprets each of the function symbols in its vocabulary. For every k-ary symbol f in the vocabulary of a state X and values $a_1, \ldots, a_k$ in its domain, some domain value b is assigned to the *location* $f(a_1, \ldots, a_k)$, for which we write $f(\bar{a}) \mapsto b$. In this way, X assigns a value $[\![t]\!]_X$ in $\mathrm{Dom}X$ to (ground) terms t.

Vocabularies are finite, since an algorithm must be describable in finite terms, so can only refer explicitly to finitely many operations. Hence, an algorithm can not, for instance, involve all of Knuth's arrow operations, $\uparrow$, $\uparrow\uparrow$, $\uparrow\uparrow\uparrow$, etc. Instead one could employ a ternary operation $\lambda x, y, z.\ x \uparrow^z y$.

This postulate is justified by the vast experience of mathematicians and scientists who have faithfully and transparently presented every kind of static mathematical or scientific reality as a logical structure.

In restricting structures to be "first-order", we are limiting the *syntax* to be first-order. This precludes states with infinitary operations, like the supremum of infinitely many objects, which would not make sense from an algorithmic point of view. This does not, however, limit the semantics of algorithms to first-order notions. The domain of states may have sequences, or sets, or other higher-order objects, in which case, the state would also need to provide operations for dealing with those objects.

Closure under isomorphism ensures that the algorithm can operate on the chosen level of abstraction. The states' internal representation of data is invisible and immaterial to the program. This means that the behavior of an *algorithm*, in contradistinction with its "implementation" as a C program—cannot, for example, depend on the memory address of some variable. If an algorithm does depend on such matters, then its full description must also include specifics of memory allocation.

It is possible to liberalize this postulate somewhat to allow the domain to grow or shrink, or for the vocabulary to be infinite or extensible, but such "enhancements" do not materially change the notion of algorithm. An extension to structures with partial operations is given in [5]; see Section 2.4.

2.2.3 Effective Transitions

The actions taken by a transition are describable in terms of updates of the form $f(\bar{a}) \mapsto b$, meaning that b is the *new* interpretation to be given by the next state to the function symbol f for values $\bar{a}$. To program such an update, one can use an assignment $f(\bar{s}) := t$ such that $[\![\bar{s}]\!]_X = \bar{a}$ and $[\![t]\!]_X = b$. We view a state X as a collection of the graphs of its operations, each point of which is a location-value pair also denoted $f(\bar{a}) \mapsto b$. Thus, we can define the *update set* $\Delta(X)$ as the changed points, $\tau(X) \setminus X$. When X is a terminal state and $\tau(X)$ is undefined, we indicate that by setting $\Delta(X) = \bot$.

The point is that Δ encapsulates the state-transition relation τ of an algorithm by providing all the information necessary to update the interpretation given by the current state. But to produce $\Delta(X)$ for a particular state X, the algorithm needs to evaluate some terms with the help of the information stored in X. The next postulate will ensure that Δ has a finite representation and its updates can be determined and performed by means of only a finite amount of work. Simply stated, there is a fixed, finite set of ground terms that determines the stepwise behavior of an algorithm.

Postulate III (Effective Transitions).[4] *For every algorithm, there is a finite set T of (ground)* critical terms *over the state vocabulary, such that states that agree on the values of the terms in T also share the same update sets. That is, $\Delta(X) = \Delta(Y)$, for any two states X, Y such that $[\![t]\!]_X = [\![t]\!]_Y$ for all $t \in T$. In particular, if one of X and Y is terminal, so is the other.*

The intuition is that an algorithm must base its actions on the values contained at locations in the current state. Unless all states undergo the same updates unconditionally, an algorithm must explore one or more values at some accessible locations in the current state before determining how to proceed. The only means that an algorithm has with which to reference locations is via terms, since the values themselves are abstract entities. If every referenced location has the same value in two states, then the behavior of the algorithm must be the same for both of those states.

This postulate—with its fixed, finite set of critical terms—precludes programs of infinite size (like an infinite table lookup) or which are input-dependent.

A careful analysis of the notion of algorithm in [62] and an examination of the intent of the founders of the field of computability in [41] demonstrate that the Sequential Postulates are in fact true of all ordinary, sequential algorithms, the (only) kind envisioned by the pioneers of the field. In other words, all *classical* algorithms satisfy Postulates I, II, and III. In this sense, the traditional notion of algorithm is precisely captured by these axioms.

Definition 1 (Classical Algorithm). *An object satisfying Postulates I, II, and III shall be called a* classical algorithm.

[4] Or **Bounded Exploration**.

2.2.4 Equivalent Algorithms

It makes sense to say that two algorithms have the same behavior, or are *behaviorally equivalent*, if they operate over the same states and have the same transition function.

Two algorithms are *syntactically equivalent* if their states are the same up to renaming of symbols (α-conversion) in their vocabularies, and if transitions are the same after renaming.

For a wide-ranging discussion of algorithm equivalence, see [4].

2.3 Abstract State Machines

Abstract state machines (ASMs) are an all-powerful description language for the classical algorithms we have been characterizing.

2.3.1 Programs

The semantics of the ASM statements, assignment, parallel composition, and conditionals, are as expected, and are formalized below. The program, as such, defines a single step, which is repeated forever or until there is no next state.

For convenience, we show only a simple form of ASMs. Bear in mind, however, that much richer languages for ASMs are given in [61] and are used in practice [64].

Programs are expressed in terms of some vocabulary. By convention, ASM programs always include symbols for the Boolean values (true and false), undef for a default, "undefined" value, standard Boolean operations ($\neg$, $\wedge$, $\vee$), and equality ($=, \neq$). The vocabulary of the sorting program, for instance, contains $\mathcal{F} = \{1, 2, +, >, F, n, i, j\}$ in addition to the standard symbols. Suppose that its states have integers and the three standard values for their domain. The nullary symbols 0 and n are fixed programming constants and serve as bounds of F. The nullary symbols i and j are programming "variables" and are used as array indices. All its states interpret the symbols $1, 2, +, >$, as well as the standard symbols, as usual. Unlike i, j, and F, these are static; their interpretation will never be changed by the program. Initial states have $n \geq 0$, $i = 0$, $j = 1$, some integer values for $F(0), \ldots, F(n-1)$, plus undef for all other points of F. This program always terminates successfully, with $j = n = i + 1$ and with the first n elements of F in nondecreasing order.

There are no hidden variables in ASMs. If some steps of an algorithm are intended

to be executed in sequence, say, then the ASM will need to keep explicit track of where in the sequence it is up to.

2.3.2 Semantics

Unlike algorithms, which are observed to either change the value of a location in the current state, or not, an ASM might "update" a location in a *trivial* way, giving it the same value it already has. Also, an ASM might designate two conflicting updates for the same location, what is called a *clash*, in which case the standard ASM semantics are to cause the run to fail (just as real-world programs might abort). An alternative semantics is to imagine a nondeterministic choice between the competing values. (Both were considered in [61].) Here, we prefer to ignore both nondeterminism and implicit failure, and tacitly presume that an ASM never involves clashes, albeit this is an undecidable property.

To take the various possibilities into account, a *proposed* update set $\Delta^+_P(X)$ (cf. [7]) for an ASM P may be defined in the following manner:

$$\begin{aligned}
\Delta^+_{f(s_1,\ldots,s_n):=t}(X) &= \{f([\![s_1]\!]_X, \ldots, [\![s_n]\!]_X) \mapsto [\![t]\!]_X\} \\
\Delta^+_{\textbf{do } \{P_1 \cdots P_n\}}(X) &= \Delta^+_{P_1}(X) \cup \cdots \cup \Delta^+_{P_n}(X) \\
\Delta^+_{\textbf{if } C \textbf{ then } P \textbf{ else } Q}(X) &= \begin{cases} \Delta^+_P(X) & \text{if } X \models C \\ \Delta^+_Q(X) & \text{otherwise} \end{cases} \\
\Delta^+_{\textbf{if } C \textbf{ then } P}(X) &= \begin{cases} \Delta^+_P(X) & \text{if } X \models C \\ \varnothing & \text{otherwise .} \end{cases}
\end{aligned}$$

Here $X \models C$ means, of course, that Boolean condition C holds true in X. When the condition C of a conditional statement does not evaluate to `true`, the statement does not contribute any updates.

When $\Delta^+(X) = \varnothing$ for ASM P, its execution halts with success, in terminal state X. (Since no confusion will arise, we are dropping the subscript P.) Otherwise, the updates are applied to X to yield the next state by replacing the values of all locations in X that are referred to in $\Delta^+(X)$. So, if the latter contains only trivial updates, P will loop forever.

For terminal states X, the update set $\Delta(X)$ is $\bot$, to signify that there is no next

	States X such that	Update set $\Delta(X)$
0	$[\![j]\!] = [\![n]\!] = [\![i]\!] + 1$	$\bot$
1	$[\![j]\!] = [\![n]\!] \neq [\![i]\!] + 1$	$i \mapsto [\![i]\!] + 1,\ j \mapsto [\![i]\!] + 2$
2	$[\![j]\!] \neq [\![n]\!]\,,\ [\![F(i)]\!] > [\![F(j)]\!]$	$F([\![i]\!]) \mapsto [\![F(j)]\!]\,,\ F([\![j]\!]) \mapsto [\![F(i)]\!]\,, j \mapsto [\![j]\!] + 1$
3	$[\![j]\!] \neq [\![n]\!]\,,\ [\![F(i)]\!] \not> [\![F(j)]\!]$	$j \mapsto [\![j]\!] + 1$

Table 2.1: Update sets for sorting program.

state. For non-terminal X, $\Delta(X)$ is the set of non-trivial updates in $\Delta^+(X)$. The update sets for the sorting program (Algorithm 1) are shown in Table 2.1, with the subscript in $[\![\cdot]\!]_X$ omitted. For example, if state X is such that $n = 2$, $i = 0$, $j = 1$, $F(0) = 1$, and $F(1) = 0$, then (per row 2) $\Delta^+(X) = \{F(0) \mapsto 0, F(1) \mapsto 1, j \mapsto 2\}$. For this X, $\Delta(X) = \Delta^+(X)$, and the next state $X' = \tau(X)$ has $i = 0$ (as before), $j = 2$, $F(0) = 0$ and $F(1) = 1$. After one more step (per row 1), in which F is unchanged, the algorithm reaches a terminal state, $X'' = \tau(X')$, with $j = n = i + 1 = 2$. Then (by row 0), $\Delta^+(X'') = \varnothing$ and $\Delta(X'') = \bot$.

2.4 The Representation Theorem

Abstract state machines clearly satisfy the three Sequential Postulates: ASMs define a state-transition function; they operate over abstract states; and they depend critically on the values of a finite set of terms appearing in the program (and on the unchanging values of parts of the state not modified by the program). For example, the critical terms for our sorting ASM are all the terms appearing in it, except for the left-hand sides of assignments, which contribute their proper subterms instead. These are $j \neq n$, $(j = n) \wedge (i + 1 \neq n)$, $F(i) > F(j)$, $i + 2$, $j + 1$, and their subterms. Only the values of these affect the computation. Thus, any ASM describes a classical algorithm over structures with the same vocabulary (similarity type).

The converse is of greater significance:

Theorem 2 (Representation [62, Theorem 6.13])**.** *Every classical algorithm, in the sense of Definition 1, has a behaviorally equivalent ASM, with the exact same states and state-transition function.*

The proof of this representation theorem constructs an ASM that contains con-

ditions involving equalities and disequalities between critical terms. Closure under isomorphisms is an essential ingredient for making it possible to express any algorithm in the language of terms.

A typical ASM models partial functions (like division or tangent) by using the special value, undef, denoting that the argument is outside the function's domain of definition, and arranging that most operations be strict, so a term involving an undefined subterm is likewise undefined. The state of such an ASM would return true when asked to evaluate an expression $c/0 = \mathsf{undef}$, and it can, therefore, be programmed to work properly, despite the partiality of division.

In [5], the analysis and representation theorem have been refined for algorithms employing truly partial operations, operations that cause an algorithm to hang when an operation is attempted outside its domain of definition (rather than return undef). The point is that there is a behaviorally equivalent ASM that never attempts to access locations in the state that are not also accessed by the given algorithm. Such partial operations are required in the next section.

Chapter 3

What is an Effective Algorithm?

3.1 Introduction

In 1900, David Hilbert posed, among other problems, the research challenge of how to effectively determine whether any given polynomial with rational coefficients has rational roots [70]:[1]

> **[Probleme] 10. Entscheidung der Lösbarkeit einer Diophantischen Gleichung.** Eine *Diophantische* Gleichung mit irgend welchen Unbekannten und mit ganzen rationalen Zahlencoefficienten sei vorgelegt: man soll ein Verfahren angeben, nach welchem sich mittelst einer endlichen Anzahl von Operationen entscheiden läßt, ob die Gleichung in ganzen rationalen Zahlen lösbar ist."

In the same lecture, as his famous second problem, Hilbert asked for a proof of the consistency of (Peano) arithmetic.

Later, he and Wilhelm Ackermann underscored the importance of the decision problem for validity of formulaæ in (first-order predicate) logic, which they called the *Entscheidungsproblem* [71, pp. 73–74]:[2]

[1] **[Problem] 10. Determination of the solvability of a Diophantine equation.** Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.

[2] The Entscheidungsproblem is solved when we know a procedure that allows for any given logical expression to decide by finitely many operations its validity or satisfiability. . . . The Entscheidungsproblem must be considered the main problem of mathematical logic. . . . The solution of the Entscheidungsproblem is of fundamental significance for the theory of all domains whose propositions could be developed on the basis of a finite number of axioms.

> Das Entscheidungsproblem ist gelöst, wenn man ein Verfahren kennt, das bei einem vorgelegten logischen Ausdruck durch endlich viele Operationen die Entscheindung über die Allgemeingültigkeit bzw. Erfüllbarkeit erlaubt. Das Entscheidungsproblem muss als das Hauptproblem der mathematischen Logik bezeichnet werden.... Die Lösung des Entscheidungsproblems ist für die Theorie aller Gebiete, deren Sätze überhaupt einer logischen Entwickelbarkeit aus endlich vielen Axiomen fähig sind, von grundsätzlicher Wichtigkeit.

Hilbert was seeking an effective procedure that could solve every instance of the validity question, positively or negatively: "We assume that we have the capacity to name things by signs, that we can recognize them again. With these signs we can then carry out operations that are analogous to those of arithmetic and that obey analogous laws" (quoted in [107]).

In 1936, Alonzo Church suggested that the recursive functions, or the computationally equivalent lambda-definable numeric functions, capture the intended concept of "effectively calculable" procedure [26, p. 356]. With his formalization of absolute effectivity in hand, he proceeded to demonstrate that no effective solution exists for the Entscheidungsproblem. When Church subsequently learned of Alan Turing's independent proof of undecidability [113], he conceded that Turing's machines have "the advantage of making the identification with effectiveness in the ordinary (not explicitly defined) sense evident immediately" [25, p. 43]. Similarly, Kurt Gödel [54, pp. 369–370] realized that Turing's model of effective computation, which provides "a precise and unquestionably adequate definition of the general concept of formal system", strengthens his earthshaking incompleteness results and establishes that "the existence of undecidable arithmetical propositions and the non-demonstrability of the consistency of a system in the same system can now be proved rigorously for every consistent formal system containing a certain amount of finitary number theory". In short, Hilbert's dream of devising a complete and consistent finite axiomatization of mathematics, as expressed in his second problem, is inherently unattainable.

Stephen Kleene reformulated Church's contention that the recursive functions and the effective numeric functions are one and the same as a "thesis" ([73, p. 60], [74, p. 332], [75, p. 232]):

> **Thesis I.** Every effectively calculable function (effectively decidable predi-

cate) is general recursive.

> **Thesis I†.** Every partial function which is effectively calculable (in the sense that there is an algorithm by which its value can be calculated for every n-tuple belonging to its range of definition) is potentially partial recursive.
>
> Turing's and Church's theses are equivalent. We shall usually refer to them both as *Church's thesis*, or in connection with that one of its... versions which deals with "Turing machines" as *the Church-Turing thesis*.

Church's thesis asserted that the recursive functions are the only numeric functions that can be effectively computed. Turing's thesis staked the analogous claim that any function on strings that can be mechanically computed can be computed, in particular, by a Turing machine. Turing showed [113, Appendix] that with a suitable interpretation of strings as numbers, his machines compute exactly the recursive functions.

Three main lines of argument have been adduced in support of this thesis ([74, p. 320], [95, pp. 18–19], [74, p. 321]):

- Despite concerted efforts, no more powerful effective computational model has been devised.
- "By means of detailed combinatorial studies, the proposed characterizations of Turing and of Kleene, as well as those of Church, Post, Markov, and certain others, were all shown to be equivalent," as have all subsequent effective models.
- Turing's analysis of "the sorts of operations which a human computer could perform, working according to preassigned instructions" showed that these can be simulated by Turing machines.

Gödel is reported [32] to have believed "that it might be possible ... to state a set of axioms which would embody the generally accepted properties of [effective calculability], and to do something on that basis". As explained by Shoenfield [103, p. 26]:

> It may seem that it is impossible to give a proof of Church's Thesis. However, this is not necessarily the case.... In other words, we can write down some axioms about computable functions which most people would agree are evidently true. It might be possible to prove Church's Thesis from such axioms.... However, despite strenuous efforts, no one has succeeded in doing this (although some interesting partial results have been obtained).

This challenge of proving the Church-Turing Thesis is number one in Richard Shore's list of "pie-in-the-sky problems" for the twenty-first century [23]. Indeed, Harvey Friedman [49] has predicted that sometime in this century, "There will be an unexpected striking discovery that any model of computation satisfying certain remarkably weak conditions must stay within the recursive sets and functions, thus providing a dramatic 'proof' of Church's Thesis."

We discuss such an axiomatization of effectiveness in Sections 2.2–3. Unlike Turing's analysis [113], and subsequent generalizations [52, 79, 80, 105, 106, 108, 109], our axioms of effective computation are, at the same time, both formal and generic. They are formal, in that they may be cast as precise mathematical statements [15, 41]; they are generic, in that they apply to computations with arbitrary states (Section 3.2) and arbitrary programmable transitions (Section 3.4).

Computability is a more general notion than recursiveness or Turing computability. Just as Turing machines provide a computational model for strings and recursive functions for the natural numbers, there are comparable notions of effectiveness for other data types, as explained in Sections 3.2 and 3.5.

Beyond that, Turing extended the notion of computability to devices provided with oracles that "magically" provide answers to questions for which there may be no effective means of providing answers. See Sections 3.3 and 3.5.

3.2 Effective States

For an algorithm to be effective, it must be possible, not only to describe transitions finitely, but also to fully describe its initial states, that starting subset of the algorithm's states containing input values. Only a state that can be described finitely may be deemed effective.

Already in 1922, Emil Post [87, pp. 427–428] noted the following about states of effective computations:

> We ... assume [symbolic representations] to be finite and we might say discrete.... Each symbolization can be considered to consist of a finite number of unanalysable parts (unanalysable from the standpoint of the symbolization) these parts having certain properties and certain relations with each other.... The ways in which these parts can be related will be assumed

> to be specified for the whole system of symbolizations.... The number of these elementary properties and relations used is finite and ... there is a certain specific finite number of elements in each relation.... The symbol-complexes are completely determined by specifying all the properties and relations of [their] parts.... Each complex of the system can be completely described [by a conjunction of relations]....

In other words, not only should states be symbolic and be represented by relational structures, but they need to be finitely representable if they are to be effective. Accordingly, we insist that effective states harbor no information beyond the means to reach domain values, plus anything that can be derived therefrom.

To handle inputs, we postulate some subset of the critical terms, namely the *input terms*, for which every possible combination of domain values occurs in some initial state, and such that all initial states agree on all terms over the vocabulary of the algorithm except these.

In general, an algorithm's domain might be uncountable—as in Gaussian elimination over the reals, but, when we speak of "effective" algorithms, we are only interested in that countable part of the domain that can be described effectively. Thus, we may as well restrict our discussion to countable domains and assume that every domain element can be described by a term in the algebra of the states of the algorithm. Furthermore, a state's operations could easily require an infinite table lookup. Thus, the initial state of an algorithm may contain ineffective infinite information, in which case the algorithm could not be deemed effective, so we need to place finiteness restrictions on the initial states of algorithms. Another problem is that the same domain element might be accessible via several terms, generating non-trivial relations, which might hide non-computable information.

Several ways of overcoming these potential problems and capturing the notion that initial states have a finite description, thereby characterizing effectiveness, have been suggested. One alternative [92] characterizes an effective (initial) state as one for which there is a (semi-) decision procedure for equality of terms in the state. That is, there is Turing machine for determining whether a state interprets two terms (given as strings) as the same domain value. A second alternative [41] requires that there exist an (arbitrary) injection from the chosen domain of the algorithm into the natural numbers such that the given base functions (in initial states) are all tracked (under that injection)

by partial-recursive functions. These two approaches are somewhat circular: the first relies on Turing-machine computability and the second on recursive functions.

A more objective approach [15] takes its cue from constructor domains. Constructors provide a way to give a unique name for any domain element, and the domain can be identified with the Herbrand universe (free-term algebra) over constructors. Destructors provide an inverse operation for constructors. For every constructor c of arity k, we may have destructors $c_1, \ldots, c_k$ to extract each of its arguments $[c_i(c(x_1, \ldots, x_i, \ldots, x_k)) = x_i]$, plus c_0, which returns an indicator that the root constructor (of a value) is c. Constructors and destructors are the usual way of thinking of domain values of effective computational models. For example, strings over an alphabet {a,b,...} are constructed from a scalar (nullary) constructor $\varepsilon()$ and unary constructors a$(\cdot)$, b$(\cdot)$, while destructors may read and remove the last letter. Natural numbers in unary (tally) notation are normally constructed from (unary) successor and (scalar) zero, with predecessor as destructor. The positive integers in binary notation are constructed out of (the scalar) ε and (unary) digits 0 and 1, with the constructed string understood as the binary number obtained by prepending the digit 1. The destructors are the last-digit and but-last-digit operations. For Lisp's nested lists (s-expressions): the constructors are (scalar) `nil` (the empty list) and (binary) `cons` (which adds an element to the head of a list); the destructors are `car` (first element of list) and `cdr` (rest of list). To construct 0-1-2 trees, we would have three constructors, $A()$, $B(\cdot)$, and $C(\cdot, \cdot)$, for nodes of out-degree 0 (leaves), 1 (unary), and 2 (binary), respectively. Destructors may choose a child subtree, and also return the degree of the last-added (root) node.

We may assume that domains include two distinct truth values and another distinct default value, and—furthermore—that we have (scalar) constructors, true, false, and undef, for all three. Boolean operations are effective finite tables, so we may presume them.

Definition 3 (Effective State [15])**.** *A state is* effective *if its domain is isomorphic to a free constructor algebra and its operations all fall into one of the following categories: those free constructors and their corresponding destructors and equality; (infinitely) defined operations that can themselves be computed effectively with those same constructors (perhaps using a richer vocabulary); and finitely many other defined location-values (not undef).*

In general, then, the operations in states come in three flavors: domain constructors; defined functions; and black-box oracles. For a state to be effective, it should provide means to access all the elements of its domain and should not have any oracles.

Function symbols C *construct* a particular domain in a given state if the state assigns each value in the domain to exactly one term over C (so the terms over C form a free Herbrand algebra). Constructors are the usual way of thinking of the domain values of computational models. For example, strings over an alphabet {a,b,...} are constructed from a nullary constructor $\varepsilon()$ and unary constructors a$(\cdot)$, b$(\cdot)$, etc. The positive integers in binary notation are constructed out of the nullary ε and unary 0 and 1, with the constructed string understood as the binary number obtained by prepending the digit 1. A domain consisting of integers and Booleans can be constructed from true, false, 0, and a "successor" function that takes non-negative integers (n) to the predecessor of their negation ($-n-1$) and negative integers ($-n$) to their absolute value (n). To construct 0-1-2 trees, we would have three constructors, $k_0()$, $k_1(\cdot)$, and $k_2(\cdot,\cdot)$, for nodes of outdegree 0 (leaves), 1 (unary), and 2 (binary), respectively.

Definition 4 (Effective State)**.**

- A state is *basic* if it includes constructors for its domain, plus totally undefined operations, meaning that they all always yield the same default value (undef, say), and no oracles.
- Such states are *(absolutely) effective.*
- Moreover, a state is effective also if all its defined operations can be effectively computed (in a bootstrapped sense to be made precise below) from basic states and with the same constructors.

Proposition 5 (Effectiveness is invariant under effecive transition)**.** *Let $\mathcal{A}$ be an algorithm with transition τ. Let X be an effective (basic) state of $\mathcal{A}$. Then $\tau(X)$ is also an effective (basic) state of $\mathcal{A}$.*

Proof. Follows from the fact that according to the Postulate III effective transition may affect only finitely many locations of X. □

This effectiveness requirement excludes states with ineffective oracles, such as the halting function, but allows one to be given effective operations, like equality of trees

or division of integers. Having only free constructors at the foundation prevents the hiding of potentially uncomputable information by means of equalities between distinct representations of the same domain element. This is the approach to effectiveness advocated in [15], extended to include partial functions in states, as in [5].

3.3 Oracular States

Turing [114] introduced the powerful idea of computability relative to oracles, saying, "We shall not go any further into the nature of this oracle apart from saying that it cannot be a machine." We may think of a Turing machine that is equipped with a special tape for querying oracles and special states q_M and q_o for each oracle o in $\mathcal{O}$. When, during an execution, the machine enters state q_o, the oracle magically answers by replacing the string x on the query tape with the value $o(x)$ and reverts to state q_M.

In the presence of oracles, we still want the domain to be constructive, or else there may be no finite way of representing inputs and outputs, but now we allow basic operations that may not be effective. Accordingly, we speak, instead, of "relative" effectiveness.

Definition 6 (Relatively Effective State)**.**

- A state is *basic* in oracles $\mathcal{O}$, if it includes constructors for its domain, totally undefined operations, plus oracles $\mathcal{O}$.
- Such states are *relatively effective*.
- Moreover, a state is relatively effective also if all its defined operations can be computed from basic states with the same constructors and oracles.

One can give an alternate characterization of effective state, one that is based on oracular Turing machines, extending a suggestion of Wolfgang Reisig [92].

Lemma 7. *A state X is effective relative to a set of oracles if and only if there is a Turing machine with the same oracles that can semi-decide the congruence induced by X. In other words, given two terms over the vocabulary of X as input, the machine returns* true *whenever both terms are defined and assigned the same values by X,* false *when both are defined but not equal, and diverges otherwise. Input and output for the machines's oracles is via constructor terms.*

The proof is along the lines of the non-oracular one in [16].

3.4 Effective Algorithms

The sequential postulates limit transitions to be effective, in the sense of being amenable to finite description, but they place no constraints on the nature of the contents of states. In particular, states may contain ineffective oracles. To preclude that and ensure that an algorithm is effective, in an absolute sense, it suffices to place limits on initial states.

Postulate IV (Initial State)**.** *Initial states of an* effective *algorithm are all (absolutely) effective (in the sense of Definition 4). Initial states of a* relatively effective *algorithm are all relatively effective (in the sense of Definition 6). Initial states of a basic algorithm are all basic (in the sense of Definition 4). In both cases, initial states over the same domain are all identical, except for input values.*

Since transitions make only finitely many changes, once initial states are effective, so are all subsequent states.

We will say that an algorithm *computes* a partial function f over a domain D if there are *input* terms such that their values in all initial states with domain D cover all possible input values. We also demand that those states otherwise agree on the values of all terms, so no information is hidden in individual states. Given values $\bar{a}$ for the input terms, the corresponding input state leads, via a sequence of transitions specified by the algorithm, to a terminal state in which the value of some designated *output* term is $f(\bar{a})$ whenever the latter is defined, and leads to an infinite computation whenever it is not.

When we spoke earlier (Definitions 4 and 6) of "bootstrapping", we meant that there is a way of programming the defined operations, using constructors and oracles, if any. And if there is any way of programming them, then there is an abstract-state-machine program that fits the bill. For example, with 0 and successor (add 1), one can program addition (+), starting from basic states, so addition my be included in the initial states of (absolutely) effective algorithms over the natural numbers. Multiplication is also effective, since there is a program for multiplication that makes use of addition.

We are requiring that all elements of an algorithm's domain be accessible via terms in initial states (inaccessible superfluous elements may be removed with no ill effect).

But note that a transition may cause accessible elements to become inaccessible in later states [92].

3.5 Relatively Effective Algorithms

Just like Turing extended his machines to incorporate oracles, the notion of recursive functions has been extended to allow oracles, for total functions by Turing [114, p. 175] and for partial ones later by Kleene [74, p. 178].

One form of this generalization is as follows: The *partial-recursive functions relative to oracles* $\mathcal{O}$ is the class of partial functions over the naturals, $\mathbb{N}$, that includes the constant zero (nullary) function, successor, all the projections, plus the operations in $\mathcal{O}$, and is closed under composition, primitive recursion, and minimization. We say that an algebra (with finitely or infinitely many partial functions) over the naturals is *recursive in* $\mathcal{O}$ if all its functions are.

Another extension of recursion theory applies it to domains other than the naturals. For this, we need the concept of "simulation" under encodings. An algebra $\mathcal{A}$ with domain D *simulates* an algebra $\mathcal{B}$ with domain E if there is an injective encoding ρ of E into D such that for every partial function g of $\mathcal{B}$ there is a partial function f of $\mathcal{A}$, such that $\rho \circ g = f \circ \rho$. A detailed discussion of simulations may be found in [12].

So, a state X over vocabulary F and arbitrary domain Dom is *computable over oracles* $\mathcal{O}$ if there is an encoding of Dom into the naturals and a recursive structure Y with domain $\mathbb{N}$ over oracles $\rho \circ o \circ \rho^{-1}$ for all $o \in \mathcal{O}$ that simulates X via ρ. An algorithm is *relatively computable* if all its initial states are computable all over the same oracle. And a model is *relatively computable* if all its algorithms are, via the same encoding and same oracle. Sans oracles, we call it *computable*. This is akin to a *computable algebra*, as in [50, 82, 90, 112], but we are not placing restrictions on the injective encoding.

Were we not to require the encoding to be an injection, we could trivially simulate everything by encoding everything by a single constant. One may ask whether the allowance of any injective encoding between the arbitrary domain and the natural numbers is sensible. But it turns out that, as long as all domain elements are reachable by ground terms, any arbitrary injective representation implies the existence of a bijection between the domain and the natural numbers [16, Lemma 1]. Hence, the initial

functions of a computable algorithm are isomorphic to some partial-recursive functions, which makes their effectiveness hard to dispute.

For example, one standard injective encoding of lists, with nullary "ε" and binary ":" as constructors, is given by $\rho(\varepsilon) = 0$ and $\rho(x : y) = 2^{\rho(x)}3^{\rho(y)}$. The standard bijective encoding is $\rho(\varepsilon) = 0$ and $\rho(x : y) = 2^{\rho(x)}(2\rho(y) + 1)$.

These two notions, effective relative to oracles and computable over oracles, are coextensional (cf. the non-oracle case proved in [15]).

Alternative An equivalent definition—along the lines of Gödel's [54] original definition of recursive equations—is to say that an algebra over domain Dom, with finitely many operations F, is computable relative to $\mathcal{O}$ if there exist constructors C for Dom and a finite set E of equations defining F. Each equation in E is of the form $f(\bar{s}) = t$, where f is a symbol for an operation in F, $\bar{s}$ is a tuple of constructor terms built from C and variables, and t is an arbitrary term built from F, C, and variables. The equations define an operation f in F relative to $\mathcal{O}$ if for all tuples $\bar{c}$ of ground constructor terms, one can deduce (by substitution of equals for equals) $E \cup \mathcal{O} \vdash f(\bar{c}) = d$ for *at most one* ground constructor term d, where $\mathcal{O}$ is now an infinite set of (ground) equations giving the (defined) values of the oracular functions in constructor terms.

For example, a computable algebra of lists with an append operation $\star$ is defined by $\varepsilon \star z = z$ and $(x : y) \star z = x \cdot (y \star z)$. With $\star$ as the (in this case, computable) oracle, one can define list reversal using just $r(\varepsilon) = \varepsilon$ and $r(x : y) = r(y) \star (x : \varepsilon)$.

Chapter 4

Universality

4.1 Introduction

Alan Turing, in his groundbreaking 1936 paper on the undecidability of the Halting Problem and the Entscheidungsproblem [113], also invented the notion of a universal machine. He explained the idea as follows:

> **The universal computing machine.** It is possible to invent a single machine which can be used to compute any computable sequence. If this machine I is supplied with a tape on the beginning of which is written the [standard description] of some computing machine M, then I will compute the same sequence as M.

There are two domains that are standard in discussions of computability: (1) finite strings over finite alphabets—for which the Turing-computable partial functions are the effective ones, and (2) the natural numbers—for which the effective functions are identified with the partial recursive ones. Davis [31] and Rogers [94] have proposed general definitions of universality for Turing machines and for partial recursive functions, respectively.

For other (countable) domains, we will adopt the analogous notion of effectiveness of a model of computation that was developed in [15] and was described in Chapter 3. Armed with the appropriate concept of generic effectiveness, we extend the notion of universality to arbitrary sets of functions over arbitrary domains (Section 4.4), while addressing oft-ignored questions of "honesty" of representations of domains (Section 4.3), pairings of elements (Section 4.5), and encodings of programs (Section 4.2).

4.2 Encodings

In the simplest case, a universal function simply "carries on its shoulders" a whole set of functions. Let Φ be some (usually, but not necessarily, countably infinite) set of unary functions over a domain D. Given a binary function ψ over D, let $\Psi = \{\psi_a \colon a \in D\}$, where $\psi_a = \lambda y.\, \psi(a, y)$, be the set of all specializations in its first parameter of ψ. We say that ψ is *universal* for Φ if $\Psi \supseteq \Phi$. This is the same as requiring there to be an *encoding* $\# : \Phi \to D$ such that ϕ computes $\psi_{\#\phi}$ for all $\phi \in \Phi$. In practice, of course, we are interested in an effective universal function ψ. By setting its parameter, the one universal function provides an effective means of computing any and all computable functions.

Unless stated otherwise, the functions we speak of may be partial or total. In particular, universal functions are partial by their very nature, and may be undefined at some points. In other words, their computation might not halt for some (or all) inputs. Equality of the two partial functions, the original one ϕ and the simulation $\psi_{\#\phi}$, means that $\phi(y) = \psi(\#\phi, y)$ for all $y \in D$, where equality (here, as well as later) is "Kleene's", so the two sides must also agree with regard to the values at which either is undefined, in which case both must be undefined.

We are placing no demands on the encoding ($\#$), only that there be some parameter value ($a = \#\phi$) for which the universal function (ψ) exhibits the identical input-output behavior as that of the given function (ϕ). The idea of universality is that one function can by itself compute a collection of functions, in an extensional sense, but not that it uses the same, or similar, means as the given programs.

In particular, nothing in our definition explicitly rules out a perverse, non-computable permutation of a standard enumeration of programs, like assigning even codes to total (universally halting) ϕ and odd codes to strictly partial ones. Such an encoding, however, would preclude a universal function ψ being effective, because, were it to be, then $\lambda i.\, \psi_{2i}$ would be an effective enumerator of programs for all the total recursive functions, an impossibility.

In point of fact, normally one is provided with a set of programs (standard descriptions of Turing machines, say) in some formalism (that is, programming language), which specifies the computed functions Φ, albeit with infinite duplication (there are always infinitely many programs with the same input-output behavior). There is no

harm in this, as then, from out point of view, $\#\phi$ is the code (e.g. the Gödel number) of any one out of the infinitely many programs that compute ϕ, which as it happens is an uncomputable coding. The related notion of an *interpreter* of one programming language in another, on the other hand, would suggest that the interpreter has an effective way of understanding the programs it is "'interpreting", in which case we would want distinct codes for distinctly behaving programs. This way or that, the above definition of universality captures the intended extensional containment: a function ψ is universal for a set of programs if its projections Ψ form a superset of the functions Φ computed by the given set of programs.

If one wants to incorporate functions of greater arity than 1, then one requires a family of universal functions, one for each arity, which we may imagine as one and the same varyadic function, and demand that $\phi(y_1, \ldots, y_\ell) = \psi(\#\phi, y_1, \ldots, y_\ell)$, for all arities ℓ, and values $y_1, \ldots, y_\ell$.

4.3 Representations

To relate sets of functions (extensional models of computation) over two different domains, C and D, we will employ a liberal notion of "simulation", with inputs mapped from C to D via an arbitrary (not explicitly effective) representation ρ and outputs by the same, or another, representation σ. For example, one typically represents a Platonic number n as used by recursive functions as a string in unary (1^n) or in binary. Conversely, a string can be viewed as a number in a base the size of the alphabet. By the same token, graphs and other data structures can be represented as strings. What we don't really want is for the representation to include non-trivial computational information, like whether the graph has a Hamiltonian cycle.

Definition 8 (Simulation of Functions)**.** *For partial functions, $g : C \rightharpoonup C$ and $h : D \rightharpoonup D$, and (injective) representations $\rho, \sigma : C \xrightarrow{1\text{-}1} D$, we write $h \sqsupseteq^\sigma_\rho g$ and say that h* simulates *g via ρ and σ, if $g = \sigma^{-1} \circ h \circ \rho$, qua partial functions, that is, if $\sigma(g(x)) = h(\rho(x))$ for all x in C, where equality is Kleene's and σ is strict (in the sense that σ of undefined is undefined).*

Definition 9 (Simulation of Models)**.** *For sets of partial functions, $G \subseteq [C \rightharpoonup C]$ and $H \subseteq [D \rightharpoonup D]$, we write $H \sqsupseteq^\sigma_\rho G$ and say that H* simulates *G via ρ and σ if for all $g \in G$ there exists $h \in H$ such that $h \sqsupseteq^\sigma_\rho g$. We say that H* simulates *G, and write*

simply $H \sqsupseteq G$, if $H \sqsupseteq_\rho^\sigma G$ for some choice of representations ρ and σ.

When ρ and σ are the identity function, $H \sqsupseteq_\rho^\sigma G$ is the superset relation, $H \supseteq G$. Simulation is transitive and reflexive.

When $\sigma = \rho$, it was shown in [14, Thm. 4.7] that ρ is necessarily effectively computable (over $C \cup D$) if there is an effective function $\widehat{s} \in H$ that simulates a *successor* function $s \in G$, that is, a function that enumerates its domain: $C = \{s^i(e) \colon i \in \mathbb{N}\}$ for some $e \in C$. (The significance of successor was noted in [102].) The analogous requirement for non-numerical domains is that its constructor functions be simulated. Even when $\sigma \neq \rho$, it turns out that both must be effectively computable, provided that—in addition—the identity function is simulated effectively.

Proposition 10. *If there are effective simulations over domain D of the constructor and identity functions of domain C, via representations ρ and σ, then ρ and σ are effectively computable for $C \cup D$.*

Proof. Consider the constructors 0 and $s(\cdot)$ of the naturals, by way of example, and let i be identity. It follows from the definitions that $i(y) = y$, $\sigma(i(x)) = \widehat{\imath}(\rho(x))$, and $\sigma(s(x)) = \widehat{s}(\rho(x))$, where the hats indicate the respective simulating functions over D. Putting those together, we have $\widehat{\imath}^{-1}(\sigma(s(x)) = \rho(s(x))$ and $\sigma(s(x)) = \widehat{s}(\rho(x))$, from which it follows that $\rho(s(x)) = \widehat{\imath}^{-1}(\widehat{s}(\rho(x)))$.

We are assuming that there are constructors for D; they can be used to effectively enumerate all of D. So, the inverse $\widehat{\imath}^{-1}$ of the simulation of the injection $i : C \xrightarrow{1\text{-}1} D$ is computable for elements of the image $\sigma(C)$ of σ in D. Hence, ρ is effectively computable, and so is $\sigma(x) = \sigma(i(x)) = \widehat{\imath}(\rho(x))$. □

When there are more arguments, the representation function ρ is extended to tuples: $\rho\langle x_1, \ldots, x_\ell\rangle = \langle\rho(x_1), \ldots, \rho(x_\ell)\rangle$. Otherwise, the above definitions are unchanged.

4.4 Universality

Using the above notion of simulation, we arrive at the following generic definition of a universal function:

Definition 11 (Universality)**.** *Let Φ be some set of unary functions (over a domain C). A binary partial function ψ (over domain D) is* universal *for Φ if Ψ $(= \{\lambda y.\, \psi(a, y) \colon a \in D\})$ simulates Φ.*

In other words, ψ is universal for Φ if there is some injective input representation $\rho : C \xrightarrow{1\text{-}1} D$, and an injective output representation $\sigma : C \xrightarrow{1\text{-}1} D$, plus an arbitrary encoding $\# : \Phi \to D$ of the functions in Φ, such that $\psi(\#\phi, \rho(x)) = \sigma(\phi(x))$ for all $\phi \in \Phi$ and $x \in C$. Note that if ψ is universal ($\Psi \sqsupseteq \Phi$) and ψ' simulates ψ (hence, $\Psi' \sqsupseteq \Psi$), then ψ' is also universal ($\Psi' \sqsupseteq \Phi$ by transitivity). Cf. [95, Thm. 1].

Our main result is that an effectively computable universal function can only simulate effective functions, as long as it simulates the constructors of a domain. Thus, the representation cannot in fact provide the universal function with any information that might allow computation of the uncomputable.

Theorem 12. *Let Φ be some set of unary functions over a domain C, including constructors and identity. Then, if there is an effective universal function (over any domain D) for Φ, then all the simulated functions $\phi \in \Phi$ are also effective.*

Proof. Let ψ be the universal function. We have $\phi = \sigma^{-1} \circ \psi_{\#\phi} \circ \rho$, for any $\phi \in \Phi$. By Proposition 10, ρ and σ are effective, since the simulations ($\widehat{s} = \psi_{\#s}$, $\widehat{\imath} = \psi_{\#i}$, etc.) of the constructors and identity (s, i, and the like) are effective. So ϕ is an effective composition of effective functions. □

In other words, the universal function cannot underhandedly simulate harder functions than it itself is capable of computing.

Nevertheless, just because ρ is effective does not mean that it cannot hide some information, albeit computable, in the representation, simply by mapping $x \in C$ to a "tuple" $[x, f(x), g(x), \dots]$ in D, for finitely many computable functions f, g, etc., such as Hamiltonianism. (The square brackets here stand for any standard tupling operation that effectively converts a sequence into a single element.) But if there is an effective injective ρ and universal function ψ, there is also an effective bijective ρ' and universal function ψ', with the latter doing all the hard work. So, restricting the notion of simulation to bijective representations, though that is not the way things are usually done, could make sense.

4.5 Pairing

One potential problem with the above definition of universal function is that some models of computation—like Turing machines—do not take their inputs separately, but,

rather, all functions are unary (string-to-string for Turing machines). In such cases, one needs to be able to represent pairs (and tuples) as single elements. One standard pairing function for the naturals is the injection $\langle i, j\rangle := 2^i 3^j$. Another, among many, is the Cantor bijection $\langle i, j\rangle := \frac{1}{2}(i + j)(i + j + 1) + j$ or this one: $\langle i, j\rangle := 2^i(2j + 1)$. For strings, one usually uses an injection like $\langle u, w\rangle := u;w$, where ";" is some symbol not in the original string alphabet.

There are several ways to go. The pairing function could reside in the source domain C, or in the target domain D, or in the representation of C as D. Regardless, this need raises a critical issue. Unless we demand that pairing be effective, there could be a machine that does too much, computing even non-effective functions. For example, a naïve definition might simply ask that pairing be injective and that $\phi(x) = \psi\langle\#\phi, x\rangle$ for all $\phi \in \Phi$ and $x \in D$ (omitting parenthesis around the pair). The problem is that an injective pairing could cheat and include the answer in the "pair". For Turing machines, say, the pair $\langle u, w\rangle$ might be represented as $u;w$ when machine u halts on input w and as $u:w$ when it doesn't. Better yet, one could map $\langle\#\phi, y\rangle \mapsto [\phi(y), \#\phi, y]$, where the square brackets are some ordinary tupling function for the domain. Then a putative universal machine could effortlessly "compute" virtually anything, computable or otherwise, just by reading the encoded input pair.

So, we clearly need for pairing to be effectively computable, as Davis and Rogers also insist. But we are talking about models in which no function takes two arguments, so we might not have an appropriate notion of computable binary function at our disposal. To capture effectiveness of pairing in such circumstances, we demand the existence of component-wise successor functions. Given a successor function s for domain D (i.e. $D = \{s^n(e)\}$ for some $e \in D$) and a pairing function $\langle\cdot,\cdot\rangle : D\times D \xrightarrow{1\text{-}1} D$, the component-wise successor functions operate as follows: $s_1 : \langle a, b\rangle \mapsto \langle s(a), b\rangle$ and $s_2 : \langle a, b\rangle \mapsto \langle a, s(b)\rangle$. If s, s_1 and s_2 are effective, then we will say that *pairing is effective*. This is because one can program pairing so that $\langle z, y\rangle := s_1^i(s_2^j\langle e, e\rangle)$, where $z = s^i(e)$ and $y = s^j(e)$. And if pairing is effective, then its two projections (inverses), $\pi_1 : \langle a, b\rangle \mapsto a$ and $\pi_2 : \langle a, b\rangle \mapsto b$, are likewise effective. (Generate all representations of pairs in a zig-zag fashion, until the desired one is located. What the projections do with non-pairs is left up in the air.)

Another concern is that requiring that pairing be computable is too liberal for the purpose. One does not really want the pairing function to do all the hard real work

itself. For example, the mapping could include $\phi(x)$ in the pair, even if it only can do that for ϕ that are known to be total (like, for the primitive recursive functions, of which there are infinitely many), or all functions that halt within some recursive bound. That would make it a trivial matter to be universal for those functions—just transcribe the answer from the input.

If, in addition to being effective, pairing is bijective, then we will deem it *honest*, since then there is no room for hiding information. For bijective pairing with effective projections, there is an effective means of forming a pair $\langle a, b\rangle$ (by enumerating all of D until the two projections give a and b, respectively). Note that, with bijectiveness alone, without effectivity, one could still hide a fair amount of uncomputable information in a bijective mapping. For instance, imagine that 0 is the code of the totality predicate and that the rest of the naturals code the partial-recursive functions in a standard order. Map pairs $(i+1, z)$ to $3\langle i, z\rangle$, where $\langle\cdot,\cdot\rangle$ is a standard pairing; map $(0, z)$ to $3j+1$ when z is the (code of the) jth total (recursive) function; and map $(0, z)$ to $3k+2$ when z is the kth non-total (partial recursive) function. Now, let U be some standard effective universal function. Then, for y divisible by 3, $\psi(y) := U(y/3)$ would compute all the partial-recursive functions, whereas $\psi(y) := y \equiv 1 \pmod 3$ would compute the uncomputable totality predicate, when $y = (0, z)$ is not divisible by 3.

In the presence of a (not necessarily honest) pairing function, we say that ψ is universal if $\Psi \sqsupseteq \Phi$, where, this time, $\psi_a = \lambda y.\, \psi\langle a, y\rangle$.

Definition 13 (Unary Universality)**.** *Let Φ be some set of unary functions (over a domain C). A unary function ψ (over domain D) is* universal *for Φ, via pairing function $\langle\cdot,\cdot\rangle$ (over D), if Ψ $(= \{\lambda y.\, \psi\langle a, y\rangle \colon a \in D\})$ simulates Φ. If, in addition, pairing is bijective, then we call the universal function* honest*.*

That is, ψ is universal if $\psi\langle \#\phi, \rho(x)\rangle = \sigma(\phi(x))$ for $\phi \in \Phi$ and $x \in C$. Of course, we are interested in the case where both pairing and the universal function are effective.

Theorem 14. *Let Φ be some set of unary functions over a domain C, including constructors and identity. Then, if there is an effective unary universal function (over any domain D) for Φ, via an effective pairing, then all the simulated functions $\phi \in \Phi$ are also effective.*

Proof. Apply Theorem 12 to $\psi'(z, y) := \psi\langle z, y\rangle$. □

Suppose $\Phi = \{\phi_z\}_z$ is some standard enumeration of (the definitions of) the partial-recursive functions. Based on Davis's second definition of a universal Turing machine (which relies on a notion of effective mappings between strings and numbers, namely, recursive in Gödel numberings), Rogers defines the property "universal(III)" of a unary numerical function ψ to be that $\phi_z(x) = \gamma(\psi\langle z, x\rangle)$ for some effective (recursive) bijection γ and effective (but perhaps dishonest) pairing $\langle\cdot,\cdot\rangle$. Let's say that such a ψ is *Rogers-universal*.

We may conclude the following from the definitions:

Theorem 15. *If a function is Rogers-universal, then it is universal in the sense of Definition 13. Furthermore, there is an honest effective universal function.*

Proof. Let ψ be the given Rogers-universal function. Take the standard enumeration for the encoding $\#$, identity for the input representation ρ, and γ^{-1} (which is well-defined and effective) for the output representation σ. Then, $\psi\langle\#\phi_z, \rho(x)\rangle = \psi\langle z, x\rangle = \sigma(\phi_z(x))$, as required.

For the second part, take some bijective pairing $\langle\!\langle\cdot,\cdot\rangle\!\rangle$ with effective projections π_1 and π_2, and let $\psi' := \lambda r.\, \psi\langle\pi_1(r), \pi_2(r)\rangle$ be our honest universal function. Putting those together, we get $\psi'\langle\!\langle z, x\rangle\!\rangle = \psi\langle z, x\rangle = \sigma(\phi_z(x))$. □

Chapter 5

Generic Evolving Systems

5.1 Introduction

As we mentioned in the Introduction, there are multiple levels at which to understand the same overall system necessitates an abstraction mechanism. Atomic physics is of no relevance to the ecologist; the ecologist's view of the system is the same regardless of whether the standard model of quantum physics is at play at the atomic level or not. This means that the behavior of the entities at the ecological level should be modeled independently of the underlying physical model, which translates into the requirement that states qua structures are isomorphism-closed (making them oblivious as to how the domain values they deal with are in fact implemented) and that their evolution respects those isomorphisms. The main goal of this chapter is to provide a well-define set of rules of such system

On each level of the ecological system, there are interacting entities: populations interact on the ecology level, organisms on the biological level, cells on the biochemical level, etc. The interacting entities need not all have "algorithmic" behavior. Aspects of the external environment (such as weather conditions) can also be treated as entities with which algorithmic components trade information. Accordingly, we need a model of communication between entities, which we shall refer to as "cells", in addition to a model of their individual evolution. To that end, we can allow the control of one cell to access values in another cell—a shared-memory viewpoint, or request values from another cell—a message-based framework. Similarly, we can allow one cell to set values in another cell or to request those changes from the other cell (depending again on one's viewpoint). We can address both the shared-memory and message-passing paradigms

in very similar manners.

Many systems, be they natural or artificial, create new entities as they evolve in time. Accordingly, we also model the "birth" of new component cells.

5.2 Formalization

We progress from the simple to the complex: cells, organs, systems. And from the general to the specific: connectivity, communication, evolution, programming.

5.2.1 Entities

We consider evolving networks of communicating entities, called *cells*. Cells are drawn from a set (or class) $\mathbb{C}$. Each cell possesses an interface comprising various *ports*, some designated for input, called *in-ports*, and others for output, *out-ports*.

The *Principle of Connectivity* is that cells communicate via directed *channels* between ports, each of which is controlled by a cell. We denotet $V = V^\iota \uplus V^o$ be the bipartite space of all possible ports, in-ports V^ι and out-ports V^o. Each port is associated with a particular cell in $\mathbb{C}$. If V_c are the ports of cell $c \in \mathbb{C}$, then V_c^ι is its in-ports, and V_c^o is its out-ports. Each port is assigned a (permanent) *position* within the cell. We may refer to a port at position p in cell c by the name $c.p$.

Channels connect two ports with each other and are used for communication between cells. Each channel is *owned* (or "controlled") by exactly *one* cell, not necessarily one that it connects to. The creator of a created (non-primeval) channel is its owner. The set of possible channels of communication is $E = V^o \times V^\iota \times \mathbb{C}$. These may be viewed as directed edges of a bipartite graph with edges from V^o to V^ι, labeled by owners in $\mathbb{C}$. An owner has the inherent right to modify or delete its channels.

Any particular system has a *network* of connections via a set of "active" channels $G \subseteq E$. A channel with in-port ι and out-port o is denoted $o \rightarrowtail \iota$. Ownership is indicated by labeling the channel: $o \overset{c}{\rightarrowtail} \iota$, or "coloring" it with the (unique) color of its owner. A channel is usually owned by one of the cells it connects. If it is owned by its out (tail) end, we call it *blue-tailed* and draw it $o \rightarrowtail \iota$. If it is owned by its in (head) end, we call it *red-headed* and draw it $o \rightarrowtail \iota$. In sum, the *topology* of a system entails cells $\mathbb{C}$, ports V, and a network that is defined by a ternary relation G involving out-ports, in-ports, and owners. One may wish to limit the number of channels that

may be connected to a port, or to in-ports only. This translates into a bound on the (total) degree, or (just) in-degree, of vertices in the graph G.

5.2.2 Interaction

The cells of a system *interact by* means of inter-cell *communication* over its channels. Communication flows *from* the cell at the out-port side *to* the side that is plugged into an in-port. So the flow is in the direction of the arrow used to display connectivity of channels. The out-port end of a channel takes output from the cell at that end; the other end, the *in-port*, provides input to the cell at that end. The ports of cells harbor values, which need not stay constant. Each channel communicates the value of its out-port to the cell at its input end. Think of the channel as a sensor wire that transmits information from surrounding cells in the environment. Communiqués within a system take on values from some fixed (finite or infinite) *alphabet* A. The *value* of port $p \in V$ is denoted $[\![v]\!]$. (Later, this notation will be refined to refer to the state of the cell in which p resides.) The idea is that the two ports at the two sides of a channel should share the same value: $[\![\iota]\!] = [\![o]\!]$ for every channel $o \rightarrowtail \iota$. Some sort of handshaking may be warranted before this equality is established. Until then, the out-port's value might be unavailable, that is, undefined. *Dormant* ports are those whose value is *undefined*, indicated by $\perp$.

Different communication standards may be contemplated: included shared memory and message passing. Taking a shared-memory perspective: The cell at the head ι of a red-headed channel $o \rightarrowtail \iota$ *pulls (reads)* the value $[\![o]\!]$ from the port o at the other end. The cell at the tail end o of a blue-tailed channel $o \rightarrowtail \iota$ *pushes (writes)* the value $[\![o]\!]$ to the cell at the other end ι. If the channel is not owned by either end, then it involves a pull and a push.

From a message-passing perspective: The cell at the head ι of a red-headed channel $o \rightarrowtail \iota$ *requests* the value $[\![o]\!]$ from the port o at the other end. The cell at the tail end o of a blue-tailed channel $o \rightarrowtail \iota$ *serves* the value $[\![o]\!]$ to the cell at the other end ι. If the channel is not owned by either end, then it involves a request and a serve. For message-passing, it would make sense if only one channel, the requesting/serving channel, were connected to an un-owned port at any given time. There could be a time delay between a request for a value from a serving cell and its receipt by the client. This would hold up execution of that part of the client process that awaits the requested

value. We do not pursue here truly partial operations, since they may be viewed as unanswered requests.

The evolution of a cell depends in part on the communications it receives at its in-ports. *Surfaces* are partial functions from ports V to the alphabet A. The set of possible surfaces is $\Sigma = [V \rightharpoonup A]$. When all the defined ports of a surface are located in the same cell or organ, the surface is called a *membrane*. Let $M \subseteq \Sigma$ be all possible membranes. A cell produces a membrane as output.

5.2.3 Evolution

The *state* of a cell evolves. Let Ξ the set of possible states in which an individual cell can be. A cell $c \in \mathbb{C}$ may still be "embryonic", in which case it is in the undefined state $\perp$. Let's refer to cell $c \in \mathbb{C}$ in state $x \in \Xi$ as the pair $c.x$. Let's refer to the value of a port $p \in V$ when cell $c \in \mathbb{C}$ is in state $x \in \Xi$ as $[\![p]\!]_x$.

States each have a *patchboard*, which is the set of channels that it currently owns. A patchboard is a subgraph of the connection graph G. If G is the current connections of a system, then

$$c.x^\sharp \;=\; \{o \overset{c}{\rightarrowtail} \iota \in G\}$$

is state $c.x$'s patchboard. Let $B \subseteq E$ be all possible patchboards.

The output membrane of a cell is the current values of its out-ports. The output membrane of state x is

$$x^o \;=\; \lambda p \in V^o.\ [\![p]\!]_x \ .$$

Every cell also has an *input membrane*, the current values of its input ports. The input membrane of state x is

$$x^\iota := \lambda p \in V^\iota.\ [\![p]\!]_x \ .$$

The state of a cell includes a *store*, in addition to its patchboard and output membrane. The set of possible stores is denoted Q. We may, therefore, view the state of a cell as being an element of $\Xi = Q \times B \times M$. Define the following projections for states $x = \langle q, b, \sigma \rangle \in \Xi$: $x^\$ = q$, $x^\sharp = b$, and $x^o = \sigma$.

Cellular evolution is governed by *plans*. Each cell has an unchanging plan θ, which determines the evolution of the state in response to input. In general, a plan θ is a state-transition relation (or multivalued function) that depends on input surfaces. To simplify the exposition, we deal only with plans that are partial functions, $\theta : \Sigma \times \Xi \rightharpoonup \Xi$

(rather than multivalued functions $\theta : \Sigma \times \Xi \rightrightarrows \Xi$). These can be indexed as $\theta_\sigma : \Xi \rightharpoonup \Xi$ for each $\sigma \in \Sigma$. Let $\Theta = [\Sigma \times \Xi \rightharpoonup \Xi]$ be all possible plans for cellular evolution.

Evolution may affect the store, patchboard, and output membrane of a cell. It is convenient to define the following:

- $\theta^{\$}(x) = \theta(x)^{\$} \in Q$ is the *updated* store;
- $\theta^{\sharp}(x) = \theta(x)^{\sharp} \in B$ is the updated patchboard;
- $\theta^{o}(x) = \theta(x)^{o} \in M$ is the updated output membrane.

Cells have independent existence, so there can be many cells in the same state and carrying identical plans.

The *Principle of Representation* states that salient aspects of states may be captured by a relational structure. Let $\mathbb{F}$ be the (finite or infinite) vocabulary and $\mathbb{D}$ be the domain of some cell. The *shape* of a cell is given by $\mathbb{F}$ and $\mathbb{D}$. Domains may be finite (*automata*), countably infinite (*machines*), or uncountable (*processes*). $\mathbb{D}$ should include the communication language A, which could include cell identities $\mathbb{C}$. A *location* in a store comprises the identity of a cell $c \in \mathbb{C}$, a function symbol $f \in \mathbb{F}$, and a sequence $\bar{d} = (d_1, \ldots, d_n)$ of values from $\mathbb{D}$ of length n, where n is the fixed *arity* of f in c.

5.2.3.1 Signals

Cells evolve over time in response to input signals. The passage of time is an independent natural process. Imagine a ticking clock with a time-stamp at its out-port. *Time* $\mathbb{T}$ is a linearly ordered set, ordered by $<$. Usually, we assume that time has a minimal element 0. A cell's evolution may begin in time with an *initial* state. A cell's evolution over time may end in a *terminal* state.

A *signal* is a partial function $u : \mathbb{T} \rightharpoonup \Sigma$ from time to surfaces Σ that is defined within some (half-open) segment $[i..j)$ of time. That is, $u(k) = \perp$ for all $k < i$ and for all $k \geq j$ ($\geq$ is of course the reflexive closure of the inverse of $<$). Note the difference between the signal being undefined outside the interval, and for there to be a defined signal within the interval that happens to be undefined at every one of its ports. Let $U = [\mathbb{T} \rightharpoonup \Sigma]$ be the set of all possible signals. The *starting* time $u_0 \in \mathbb{T}$ of a signal $u \in U$ operating over an interval $[i..j)$ is i; the *ending* time $u_\infty \in \mathbb{T}$ is j. In other words, $u_0 = \min k \in \mathbb{T}.\ u(k) \neq \perp$ and $u_\infty = \min k > u_0.\ u(k) = \perp$.

Suppose that $\theta \in \Theta$ is the plan of cell $c \in \mathbb{C}$. It is convenient to write x_u to abbreviate $\theta_u(x)$, for $x \in \Xi$ and $u \in U$. We can write k as short for the unadulterated *time* signal $\lambda j \in [0..k).\ \lambda p \in V^\iota.\ j$, which just tracks time. Then x_k is the state that results from letting state $x \in \Xi$ evolve until time $k \in \mathbb{T}$.

Signals $u, v \in U$ may be *concatenated*, but only *provided* $v_0 = u_\infty$:

$$u \bullet v \ = \ \lambda k. \begin{cases} u(k) & \text{if } k < u_\infty \\ v(k) & \text{if } v_0 \leq k \\ \bot & v_0 \neq u_\infty . \end{cases}$$

Of course $(u \bullet v)_0 = u_0$ and $(u \bullet v)_\infty = v_\infty$.

The *Principle of Timelessness* is that signals that are alike—except for the time at which they occur—have the same effect. That is, for all plans $\theta \in \Theta$, signals $u, v \in U$, and order-preserving isomorphisms $\alpha : U \leftrightarrow v$, we have

$$u \cong_\alpha v \ \Rightarrow \ \theta_u = \theta_v \ .$$

This means that cellular functions are oblivious to the time at which a signal is received, unless—of course—the signal includes the time.

The *Principle of Causality* states that the present and the future depend only on the past. Formally, this is

$$\theta_{u \bullet v} \ = \ \theta_u \circ \theta_v \ ,$$

for plans $\theta \in \Theta$ and signals $u, v \in U$. *We are using relational notation for composition of functions*: $(\theta_u \circ \theta_v)(x) = \theta_v(\theta_u(x))$. Put differently, Causality says that

$$x_{u \bullet v} \ = \ (x_u)_v \ ,$$

for states $x \in \Xi$.

Evolving cells produce output signals in response to input signals. Let

$$u_{..j}(k) \ = \ \begin{cases} u(k) & \text{if } k < j \\ \bot & \text{if } k \geq j \ , \end{cases}$$

for $j \in [u_0..u_\infty]$, be the prefix of signal u restricted to the interval $[u_0..j)$. Then, let

$$\tilde{\theta}_u(x) \;=\; \lambda j.\, \theta^o_{u_{..j}}(x)$$

denote the *output signal* that results from applying plan θ with signal $u \in U$ to state $x \in \Xi$. It follows that

$$\tilde{\theta}_{u \bullet v} \;=\; \tilde{\theta}_u \bullet (\theta_u \circ \tilde{\theta}_v) \;.$$

Abbreviating an output signal $\tilde{\theta}_u(x)$ as $x_{\tilde{u}}$, we have that

$$x_{u \tilde{\bullet} v} \;=\; x_{\tilde{u}} \bullet (x_u)_{\tilde{v}} \;.$$

Let L be the set of locations of some cell shape. A *store* is a set of values of locations, as for classical algorithms. A cell's in-ports are among its locations. Out-ports are just those locations whose values are made available for outgoing communications. The changes in a state over time may be captured by changes to its components. To keep the notation below relatively simple, it is nice to define *anti-elements* for sets. For any set S, let $S^{-1} = \{s^{-1} : s \in S\}$ be the set of anti-elements for S. The idea is that anti-elements annihilate their counterparts: $S' \cup S^{-1} = S' \setminus S$. (So this union is not necessarily commutative.) Viewing stores as location-value pairs, define—for fixed plan $\theta \in \Theta$ and signal $u \in U$:

$$\Delta^{\$}(x) \;=\; \left(x^{\$} \setminus \theta^{\$}_u(x)\right)^{-1} \cup \left(\theta^{\$}_u(x) \setminus x^{\$}\right) \;.$$

These are anti-elements for what θ removes from x's store plus (ordinary) elements for what it adds. Viewing patchboards as port-port edges (pairs):

$$\Delta^{\sharp}(x) \;=\; \left(x^{\sharp} \setminus \theta^{\sharp}_u(x)\right)^{-1} \cup \left(\theta^{\sharp}_u(x) \setminus x^{\sharp}\right) \;.$$

Viewing the output membrane as a set of port-value pairs:

$$\Delta^{o}(x) \;=\; \left(x^{o} \setminus \theta^{o}_u(x)\right)^{-1} \cup \left(\theta^{o}_u(x) \setminus x^{o}\right) \;.$$

Bagging everything together, and keeping elements of different sorts separate, we

can let

$$\Delta(x) \;=\; \Delta^{\$}(x) \uplus \Delta^{\sharp}(x) \uplus \Delta^{o}(x) \;.$$

Then (using elements and anti-elements) a transition θ can be reconstructed from the difference Δ:

$$\theta_u(x) \;=\; x \cup \Delta(x) \;.$$

For ordinary discrete time, $\mathbb{T} = \mathbb{N}$, instead of talking about an input *signal* u, we can talk about an input *sequence*, which is a sequence of membranes. The input sequence given by a signal u: is

$$u(u_0)\; u(u_0+1)\; \ldots\; u(u_\infty - 1)$$

Let's denote this sequence of membranes

$$\sigma_0\; \sigma_1\; \ldots\; \sigma_n$$

Let σ be one of these input membranes σ_i and let Domσ be its domain of definition. The current set of values of those in-ports of a cell x may be defined in this way:

$$\sigma_* \;=\; \{p \mapsto [\![p]\!]_x : p \in \mathrm{Dom}\sigma\} \;.$$

Viewing an input membrane as a set of port-value pairs, the effect of an input step may be described as follows:

$$x^\sigma \;=\; x \cup \sigma_*^{-1} \cup \sigma \;.$$

Let $x_\sigma = \theta_{\{j \mapsto \sigma\}}(x)$ be the *next* state for $x \in \Xi$ after receiving a single-point signal $\sigma \in \Sigma$. The total effect on a cell x's store of one discrete step with input σ is:

$$x_\sigma \;=\; \theta(x^\sigma) \;.$$

In other words, first the input values are inserted into cellular locations, and then the state transitions to the next state.

5.2.4 Systems

States may come in a variety of *types*. The cellular structure of cells of the same type are of the same shape and cells of the same type are governed by the same plan. The number of types is typically finite. A *tissue* is a set of interconnected cells of the same type.

Cells can conceive and give birth to new cells of the same type. The *Principle of Motherhood* asserts that every cell has at most one mother and that mothers are aware of the identity of their offspring. Motherless cells are *primeval*.

An *organ* is either a simple cell or it is a *complex* organ consisting of a set of interacting organs. An organ may be viewed from the outside just like a cell. The *status* $X : \mathbb{C} \rightharpoonup \Xi$ of an *organ* is a partial function giving the state in which of its cells is. The set of possible organ states is $\Lambda = [\mathbb{C} \rightharpoonup \Xi]$. An organ may interact with other organs. It follows (from the fact that every channel has a unique owner) that the union of patchboards of all cells of a system with status X defines the system's network G:

$$G(X) \;=\; \bigcup\nolimits_{c \in \mathbb{C}} X(c)^\sharp \;.$$

A *system* is an organ that does not interact with any external organs. The *state* of a complex organ is the is the set of states of its internal organs, including the graph of their interconnections. An organ's ports are the union of the ports of its internal organs. The patchboard of an organ is its externally connected owned ports. The behavior of a complex organ is the sum of the behaviors of its components, its *internal organs*, which are either cells or other organs. An organ's *policy* is that each of its internal organs minds its own policy. The set of all policies is $\Pi = [\mathbb{C} \rightharpoonup [\Xi \rightharpoonup \Xi]]$. Surfaces may be *fused* to operate over the *symmetric difference* of their interfaces:

$$\sigma \ominus \sigma' = \lambda m. \begin{cases} \sigma(m) & \text{if } m \in \text{Dom } \sigma \setminus \text{Dom } \sigma' \\ \sigma'(m) & \text{if } m \in \text{Dom } \sigma' \setminus \text{Dom } \sigma \\ \bot & \text{otherwise .} \end{cases}$$

The graph $G(X)$ of an organ X induces an (unlabeled) *acquaintance* graph K based

solely on ownership:

$$K \;=\; \{c \longrightarrow a,\; c \longrightarrow b : a.p \overset{c}{\rightarrowtail} b.q \in G\} \;.$$

The idea is that a cell "knows" the cells it connects by means of a channel it owns. Let K^* be the reflexive-transitive closure of this (binary) relation K. States can only learn the identity of cells that they know about.

Plans may follow rules or not. The *Principle of Describability* is that an organ is *algorithmic* if its evolution has a finite description (in terms of changes to the structure of states). This is analogous to the case of classical algorithms. Algorithmic plans may be described by *programs*. Programs operating in *sequential* time must define the one-step transition relation θ_σ. This may be done in the basic language of abstract state machines, which includes the following at a minimum:

- general assignments: $f(s_1, \ldots, s_n) := t$ (terms $s_1, \ldots, s_n, t$ in the vocabulary of the state);
- conditionals: $r : P$ (Boolean term r and program P); and
- parallel composition: $P \& Q$ (programs P, Q).

The *semantics* of a program is a function form states to sets of updates. The semantics of the above constructs are as follows:

- general assignment:

$$[\![f(\bar{s}) := t]\!]\,(x) \;=\; \{f([\![\bar{s}]\!]_x) \mapsto [\![f([\![\bar{s}]\!])]\!]_x\}^{-1} \cup \{f([\![\bar{s}]\!]_x) \mapsto [\![t]\!]_x\} \;;$$

- conditionals:

$$[\![c : P]\!]\,(x) \;=\; \begin{cases} [\![P]\!]\,(x) & \text{if } [\![c]\!]_x = \mathsf{true} \\ \varnothing & \text{otherwise} \;; \end{cases}$$

- parallel composition:

$$[\![P \& Q]\!]\,(x) \;=\; [\![P]\!]\,(x) \cup [\![Q]\!]\,(x) \;.$$

In addition, we want:

- higher-order assignments: $f := g$; where f and g are functions (of the state vocabulary) of the same arity,
- serial composition: $P\,;Q$ (programs P, Q); and
- parallel choice: $P \mid Q$ (programs P, Q).

The semantics are as follows:

- higher-order assignments:

$$[\![f := g]\!]\,(x) \;=\; \{f(\bar{d}) \mapsto [\![f(\bar{d})]\!]_x : \bar{d} \in \mathbb{D}^*\} \cup \{f(\bar{d}) \mapsto [\![g(\bar{d})]\!]_x : \bar{d} \in \mathbb{D}^*\}\ ;$$

- serial composition:

$$[\![P\,;Q]\!]\,(x) \;=\; [\![P]\!]\,(x) \cup [\![Q]\!]\,(x \cup [\![P]\!]\,(x))\ ;$$

- parallel choice:

$$[\![P \mid Q]\!]\,(x) \text{ is either } [\![P]\!]\,(x) \text{ or } [\![Q]\!]\,(x)\ .$$

We should also allow conditional and choice terms:

- conditional: the value $[\![r : s]\!]_x$ of the conditional term $r : s$ is $[\![s]\!]_x$ provided $[\![r]\!]_x = \mathsf{true}$, and has no value otherwise;
- choice: the value $[\![s \mid t]\!]_x$ of the choice term $s \mid t$ is either $[\![s]\!]_x$ or $[\![t]\!]_x$.

Conditionals go hand-in-hand with choice, as in

$$[\![(r : s) \mid (\neg r : t)]\!]_x \;=\; \begin{cases} [\![s]\!]_x & \text{if } [\![r]\!]_x \\ [\![t]\!]_x & \text{otherwise}\ . \end{cases}$$

A *foreign* location is indicated by a port-valued expression of the form $c.f(s_1, \ldots, s_n)$, where c is cell-valued and $f(s_1, \ldots, s_n)$ yields a location in c. Only local locations and shared resources may appear on the left of assignments. A foreign resource on the left of an assignment is an *agent*. A resource that only appears on the right side or in conditions is an *asset* (that is, read-only).

A new cell may be conceived by means of a creation term

$$\nu(P)$$

where P is a program. All expressions in P are evaluated with respect to the mother, but all assignments are made to locations in the fetus (baby cell). All other operators in the baby are completely undefined. The value of this term is the identity $c \in \mathbb{C}$ of the baby. Additional assignments may be made to operations in (unborn) fetuses by referring to its locations with terms like $c.f(s_1, \ldots, s_n)$. Babies are *birthed* by means of the command

$$\&(c_1, \ldots, c_n)$$

where the $c_i \in \mathbb{C}$ are the babies' identities.

When born, the baby cell will run the same algorithm as its mother, but its store may differ. Flags can be set to specialize the behavior of offspring. The command $\&(\nu())$ launches an independent new cell with pristine store, but it would have no way to interact with other cells.

The program of an organ as a whole is just the union of the programs of its constituent "organelles" (sub-organs), with functions disambiguated by the name of the cell they reside in. Of course, some cells might not be governed by programs, but rather provide environmental input, like measurements of natural phenomena (e.g. thermometers, barometers, velocimeter). Whereas an individual *programmed* cell may own only a bounded number of channels, an organ can create more and more new cells, each of which is connected to external cells. Programs as described above can cause conflicts ("clashes") when different assignments (in one or more cell plans) attempt to assign different values (at one and the same moment) to a single location. The outcome of such a conflict is any one of the possibilities. (These nondeterministic semantics are preferable to a system crash.)

5.2.4.1 Effectiveness

The *Principle of Effectiveness* is that an algorithmic organ is *effective* if its initial status has a finite description. In general, for a system to be deemed *effective*, not only should its transitions and evolutions be describable by a finite text, but also the initial states with the operations they are endowed with. For an organ to be effective, it should have

finitely many cells, each governed by an effective algorithm. The number of cells and their inter-connections may grow unboundedly during its evolution. This is analogous to the classical case.

5.2.4.2 Continous Time

For continuous-time systems, the discrete programming language is extended with

- continuous (explicit) assignments: $f(s_1, \ldots, s_n) :\approx t$,

which stay in force until a new assignment is made to the same *term* by some program. Jumps are effected by conditionals. Additional constraints on algorithmic evolution make sense in the continuous context. These include that tests should test for conditions that have non-zero duration and that the dynamics of a system change only finitely often in a finite period of time. A *jump* in the evolution of a continuous-time cell is a change in its dynamics, in contrast with *flows*, during which the dynamics are fixed. Various other possibilities make sense for programming continuous-time systems (implicit equations, infinitesimals). To achieve synchronous behavior in a continuous-time environment, there would need to be a global *clock* to which other cells are connected, directly or indirectly.

5.2.5 Discussions

Many instances of plans come to mind. When foreign locations provide only read-only resources, write abilities to a public (but not shared) memory need to be achieved via requests—as in modern hardware. A cell c can allocate resources for requests r, addresses a, and values v, which it makes available to a memory module. The latter runs a program of the sort $p.r : m(p.a) := p.v$, for some "archive" function m. A similar setup may be used to serve stored values.

When unboundedly many cells use the same controlled archive, some queueing mechanism needs to be set up, by means of which individual cells can place requests while the archive deals with them one at a time.

If (automata) cells share a finite domain (as in cellular models), then unbounded memory is achievable by means an unbounded number of connected cells. In this case an unbounded number of steps may be needed to access a particular datum.

To model a physical or biological system in which units are each governed by rules, but adjacent units exchange values or signals, one could represent their interface as a channel. For example, the temperature of a wall would be a public function over $\mathbb{R}^2$ of one side or the other.

Positions in space (of physical or biological systems) may be modeled by having each cell keep track of its own position. Neighboring cells would need to be in contact to avoid overlap.

Chapter 6

What is a Parallel Algorithm?

6.1 Informal View

Informally, a *parallel algorithm* consists of a (finite or infinite) set of cells, whose individual *states* all evolve according to the same algorithm. The state of each *cell*, at any moment, is a (logical) structure with a tripartite vocabulary $F \uplus F' \uplus G$ consisting of *private (internal)* operations F, *public (global)* G, and *embryonic* F', the latter having the same similarity type as F. (There could be any fixed number of embryonic copies $F', F'', \ldots, F^{(k)}$, but let us leave it simple for now, one child at a time.) The individual cells all run a "classical" (sequential) algorithm in the sense of Chapter 2.

Initially, all cells agree on G and their F' are pristine (completely undefined). A single *global step* of the algorithm comprises of the following stages.

1. First, each cell C takes one classical step, producing a set of *updates* U.

2. Cells' private operations F and embryonic operations F' are updated per U.

3. Then the union of all the cell's public updates together are applied to every cell's public G. If there is any disagreement between cells regarding updates to G (the same location getting contradictory new values), the whole system aborts. (Abortion could be replaced with nondeterministic behavior, should one prefer.)

4. Assuming there are no conflicts, *mitosis* takes place as follows: Each cell C in which the values of the operations F' were modified splits into two, a mother C and daughter C'. The daughter C' inherits G, as updated, from her mother; her F is a copy of her mother's F'. For both mother and daughter, F' is reinitialized to undef

(*undefined*).

5. If one wishes, an individual cell can be allowed to *die* and be dropped from the global organism whenever it has no next state, as when it suffers an internal clash.

6.2 Parallel Algorithms

An algorithm is generally viewed as a state-transition system composed of a set of states and a transition function (or, more generally, a relation) over states.

Definition 16 (Parallel System)**.** *A* parallel system *consists of a set (or class)* $\mathcal{S}$ *of states, a (nonempty) subset* $\mathcal{S}^0 \subseteq \mathcal{S}$ *of which are* initial*, a* federacy *of (countably or uncountably many)* identities $\mathcal{I}$*,* localized *states* $X_i \in \mathcal{S}$ *for each* $X \in \mathcal{S}$ *and* $i \in \mathcal{I}$*, and a (partial)* transition *function* $\tau : \mathcal{S} \rightharpoonup \mathcal{S}$*. Whenever* τ *is undefined for a state* $X \in \mathcal{S}$*, we will say that* X *is* terminal*.*

We first explain what the states of a parallel algorithm look like and then discuss algorithmic transitions.

6.2.1 Global States

We need for systems to comprise multiple local processes, what we called "cells". As explained above, states should be formalized as (first-order) logical structures over some vocabulary, fixed by the algorithm. Since we are dealing with parallel algorithms, with both private and shared memory, each cell has a *local* state, which is a structure over a (finite) vocabulary $G \uplus F$, where the (current) values of operations in G are stored (conceptually, at least) in *global* locations, accessible to all cells, while private data is stored as values of operations in its personal copy of F.

Each cell has its own unique identity, taken from some index set (or class) $\mathcal{I}$. Suppose $F = \{f^1, \ldots, f^k\}$. Then the k local functions of cell $i \in \mathcal{I}$ are indexed $F_i = \{f_i^1, \ldots, f_i^k\}$, where, for each $j = 1, \ldots, k$, the functions f_i^j have the same arity for all cells i. It will be convenient in what follows to denote $F^j = \{f_i^j : i \in \mathcal{I}\}$. The *global* state of the algorithm will be an algebra over the combined (possibly infinite) vocabulary $V = G \cup F^*$, where $F^* = \bigcup_{i \in \mathcal{I}} F_i = \bigcup_{j=1}^{k} F^j$.

Let X be a state of $\mathcal{A}$ with domain (universe, carrier) D. Let g be some function in V (either in G or in F^*) of arity n and $\bar{u}$ be an n-tuple of elements of that domain.

Then, by interpreting g, X determines the value $[\![g]\!]_X(\bar{u})$ of its *location* $g(\bar{u})$. For any ground term t, we write $[\![t]\!]_X = w$ to mean that the value of t as interpreted in X is w.

When state X with transition τ is not terminal, we say that $g(\bar{u}) \mapsto w$ is an *update* of X if τ *changes* the value of $g(\bar{u})$ to be w in $\tau(X)$, which was not its value in X. By $\Delta_\tau(X)$, we denote the set of all updates of X under τ.

Each cell works with only part of the global state. We define the ith *localization* X_i of global state X of federacy $\mathcal{I}$ to be the structure X with its "active" vocabulary restricted to V_i, meaning that all other functions $(F^* \setminus F_i)$ are everywhere undefined, taking on the otherwise unused value undef. Note that the localization of a localization X_i is X_i itself. The evolution of the ith cell should utilize only the values of the defined functions of the ith localization, identifying its private F with the indexed functions F_i. Transitions for a cell can only change values of its functions and of its progeny. We say that a localization X_i is *empty* if every function in F_i is also undefined. These are nascent cells, yet to be born. We call X_i an *i-state* when it's not empty.

Identities are just a fiction to distinguish one cell from another; when comparing states, the individual identities should be ignored. Two states are the same if they are the same up to permutation of cell identities. Similarly, two sequences of state transitions are the same if there is a permutation of identities for the states in one of the sequences that makes it identical to the other sequence.

To facilitate state comparison, we define a *anonymization* operator $^\sharp$ that wipes off the identifier, that is, it erases the identity-index from function symbols. Thus, the anonymized $X_i^\sharp$ is obtained from a localized cell X_i by restricting the vocabulary to V_i and pretending that the remaining symbols f_i^j are just f^j, for all j. Accordingly, we say that $X_i = Y_k$, for two localizations, if $X_i^\sharp = Y_k^\sharp$, that is, if they are identical when anonymized. Similarly, we say that transition τ generates the same updates for X_i and Y_k if the updates to G are the same in both X_i and Y_k, the updates to F_i in X_i are the same as the updates to F_k in Y_k, and the updates to other locations are the same up to the choice of indices for updates to daughter cells. In such a case, we will simply write $\Delta_\tau(X_i) = \Delta_\tau(Y_k)$. We denote by $\Delta_\tau^i(X)$ the set of all updates to locations of F_i in X. As before, we write $\Delta_\tau^i(X) = \Delta_\tau^k(Y)$ if $\Delta_\tau^i(X)^\sharp = \Delta_\tau^k(Y)^\sharp$.

To capture the uniform behavior of cells, we introduce *templates*, which are terms over an unadorned vocabulary $G \cup \{f^1, \ldots, f^k\}$, where f^j is a symbol of the same arity as the $f_i^j \in F$. For each $i \in \mathcal{I}$, the template t induces a *critical* term t_i, obtained by

replacing each occurrence of f^j by f_i^j. Given states X and Y from the same transition system, we say that they *agree* on a set of templates T and indicate $X \equiv_T Y$ if $[\![t_i]\!]_X = [\![t_i]\!]_Y$ for every $t \in T$ and $i \in \mathcal{I}$. In words, every localized term defined by a template t has the same value in both X and Y.

To compare different cells we should again ignore their specific identities. Let X_i and X_k be distinct localizations of global state X. We say that $X_i \equiv_T X_k$ (the two states "agree") if $[\![t_i]\!]_X = [\![t_k]\!]_X$ for each $t \in T$. Similarly, we may compare localizations of two distinct global states. We write $X_i \equiv_T Y_k$, for localization X_i of X and Y_k of Y, if $[\![t_i]\!]_X = [\![t_k]\!]_Y$ for each $t \in T$.

6.2.2 Algorithms

Let $\mathcal{A} = (\mathcal{S}, \mathcal{S}_0, \mathcal{I}, \tau)$ be a parallel system of federacy $\mathcal{I}$ over vocabulary V. We deem it to be "algorithmic" if it satisfies a number of postulates, which we now proceed to explicate.

Postulate V (Abstract State)**.** *A parallel system is* abstract *if the following properties hold:*

1. *Its states are structures over a vocabulary V.*
2. *All states share the same vocabulary.*
3. *The functions in its states are all* strict*: $f(\ldots, \mathsf{undef}, \ldots) = \mathsf{undef}$ for all $f \in V$.*
4. *Its states (and also the set of initial states and the set of terminal states) are closed under isomorphism (of first-order structures).*
5. *Its states are also closed under localizations, that is, if X is a state, then X_i is also a state, for each identity i.*
6. *Isomorphic states are either both terminal or else their next states are isomorphic, via the same isomorphism.*
7. *Transitions preserve the domain of states.*

States as structures make it possible to consider any data structure sans encodings. In this sense, algorithms are generic. The structures are "first-order" in syntax, though domains may include sequences, or sets, or other higher-order objects, in which case the

state would provide operations for dealing with those objects. States with infinitary operations, like the supremum of infinitely many objects, are precluded. Closure under isomorphism ensures that the algorithm can operate on the chosen level of abstraction and that states' internal representation of data is invisible to the algorithm. This means that the behavior of an *algorithm*, in contradistinction with its "implementation" as a program in some particular programming language, cannot depend on the memory address of some variable.

It must be possible to describe the effect of transitions in terms of the information in the current state. To that end, we use *templates*, which refer to global locations in the current state and to local locations in each cell. Parallel algorithms use these templates to describe state transitions, without referring to cells individually. If every referenced location has the same value in two states, then the behavior of the algorithm must be the same for both of those states. This, the essence of what makes a process algorithmic, is a crucial insight of [62].

Each cell is fully responsible for its local updates. The updates created by an individual cell may not depend on its identity, but only on global and local locations that are available to it.

Postulate VI (Localization)**.** *An abstract parallel system with state-transition τ and identities $\mathcal{I}$ is* localized *if there exists a finite set T of templates such that $\Delta_\tau(X_i) = \Delta_\tau(Y_k)$ and $\Delta^i_\tau(X) = \Delta^k_\tau(Y)$ whenever $X_i \equiv_T Y_k$ for states X and Y and identities $i, k \in \mathcal{I}$.*

Moreover, all updates of a global state are generated solely by local cells.

Postulate VII (Globalization)**.** *An abstract parallel system with state-transition τ and identities $\mathcal{I}$ is* globalized *if $\Delta_\tau(X) = \bigcup_{i\in\mathcal{I}} \Delta_\tau(X_i)$ for all states X and identities $i \in \mathcal{I}$.*

For ordinary (non-parallel) algorithms, one asks for the following [62, Bounded Exploration Postulate]:

Definition 17 (Algorithmicity)**.** *An abstract parallel system with state-transition τ is* algorithmic *if there exists a finite set T of templates such that $X \equiv_T Y$ implies $\Delta_\tau(X) = \Delta_\tau(Y)$ for all states X and Y.*

Proposition 18. ***Algorithmicity*** *follows from* ***localization*** *and* ***globalization****.*

Proof. Suppose $X \equiv_T Y$ for states X, Y and some finite set of templates T. Assume that $\delta = f(u_1, \ldots, u_n) \mapsto u_0$ is an update of X. Then, by **globalization**, there exists a cell i such that δ is an update of X_i. Since $X \equiv_T Y$ there exists a j such that $X_i \equiv_T Y_j$. From **localization**, we deduce that δ is an update of Y_j, and from **globalization** that it is an update of Y. Hence $\Delta_\tau(X) \subseteq \Delta_\tau(Y)$. Inclusion of $\Delta_\tau(Y)$ in $\Delta_\tau(X)$ is proved in the same way. Hence $\Delta_\tau(X) = \Delta_\tau(Y)$, as required. □

6.2.3 Childhood

If some localization of X is empty but is not empty for $\tau(X)$, this indicates that a child has been born. We demand that once a cell has been created, no other cell can change its internals.

Postulate VIII (Fertility)**.** *An abstract parallel system with state-transition τ and identities $\mathcal{I}$ is* fertile *if there exists a (input-independent) bound $n \in \mathbb{N}$, such that, for any localization X_i of a state X with identity $i \in \mathcal{I}$, $\tau(X_i)$ has at most n non-empty localizations.*

The idea here is that in a single step each cell may participate in the creation of only a bounded number of new processes.

Lastly, each newborn cell has exactly one mother:

Postulate IX (Motherhood)**.** *An abstract parallel system with state-transition τ and identities $\mathcal{I}$ is* maternal *if whenever a localization X_i of a state X with identity $i \in \mathcal{I}$ is empty, but is non-empty for $\tau(X)$, there is an identity $k \in \mathcal{I}$ such that $\Delta^i_\tau(X) \subseteq \Delta_\tau(X_k)$.*

6.2.4 Parallel Algorithms

With the above requirements in place, we can state what a parallel algorithm is.

Definition 19 (Parallel Algorithm)**.** *A* parallel algorithm *is a parallel state-transition system that satisfies Postulates V–IX.*

Proposition 20. *Any parallel algorithm over a* finite *vocabulary may be described by an ordinary algorithm.*

Proof. Consider an algorithm over a *finite* vocabulary V. Then instead of $V = G \cup F^*$, we may assume that we only have $V = G$ (and we required that G be finite). So for

this case, the algorithm is only required to have the **abstract state** and **algorithmic** properties, and postulates VI–IX are redundant. Also, our final set of templates T is just a finite set of terms over $V = G$. So what we have is a *classical (sequential) algorithm* with *critical terms* T, as defined by Gurevich in [62]. □

6.3 Parallel Programs

The two basic program instructions are assignment and creation.

Assignment. An *atomic assignment* is an instruction of the form $g(t^1, \ldots, t^n) := t^0$, where $t^0, \ldots, t^n$ are templates and $g \in V$ has arity n.

Let X_i be a localization of X. Assume that $\left[\!\left[t_i^j\right]\!\right]_X = u_i^j$ for each $j = 0, \ldots, n$. If $g \in G$, then application of an assignment a on X for i generates an update $\Delta_a(X_i) = \{g(u_i^1, \ldots, u_i^n) \mapsto u_i^0\}$, with the appropriate index i. If $g \in F$, then the application generates an denoted $\Delta_a(X_i) = \{f_i(u_i^1, \ldots, u_i^n) \mapsto u_i^0\}$, with the appropriate index i. If one of the t^j is undefined (undef) in X_i, then $\Delta_a(X_i) = \varnothing$. The application of a to X generates the update set $\Delta_a(X) = \bigcup_{i \in \mathcal{I}} \Delta_a(X_i)$.

Parallel assignment. More generally, a *parallel assignment rule* a is a finite set of atomic assignments, written out as $a_1 \parallel a_2 \parallel \cdots \parallel a_\ell$. The update set generated by this instruction is $\Delta_a(X) = \bigcup_{j=1}^{\ell} \Delta_{a_j}(X)$. If $\Delta_a(X)$ includes conflicting updates (different values assigned to the same location), then the rule is not applied.

Creation. This is an instruction denoted **new**.a of the form **new** a, where a is a parallel assignment.

Suppose a is a single assignment $f(t^1, \ldots, t^n) := t^0$, and let X_i be one localization. The transition initializes some empty localization X_{k_i} by setting $f_{k_i}(\left[\!\left[t_i^1\right]\!\right]_X, \ldots, \left[\!\left[t_i^n\right]\!\right]_X)$ to be $\left[\!\left[t_i^0\right]\!\right]_X$. Then $\Delta_{\mathbf{new}.a}(X_i) = \{f_{k_i}(u_i^1, \ldots, u_i^n) \mapsto u_i^0\}$, where each $u_i^j = \left[\!\left[t_i^j\right]\!\right]_X$. However, if any one of the u_i^j is undefined, then $\Delta_{\mathbf{new}.a}(X_i) = \varnothing$. For each cell i, the transition chooses a different daughter cell k_i. In general, the update is appended to the total set of updates $\Delta_{\mathbf{new}.a}(X) = \bigcup_{i \in \mathcal{I}} \Delta_{\mathbf{new}.a}(X_i)$.

If a is a parallel assignment $a_1 \parallel a_2 \parallel \ldots \parallel a_\ell$, then application of **new**.a chooses a unique empty X_{k_i} for each X_i such that all the templates in the atomic assignments are defined for X_i. In this case, $\Delta_{\mathbf{new}.a}(X) = \bigcup_{j=1}^{\ell} \Delta_{\mathbf{new}.a_j}(X)$. If there is no way to

choose a unique k_i for each i such that the rule can be applied to it, then the rule is not applied at all.

Guard. An *atomic guard* is a condition of the form $s = t$ or $s \neq t$. A guard $t = s$ evaluates to true for localization X_i if $[\![t_i]\!]_X = [\![s_i]\!]_X$, and $t \neq s$ is true if $[\![t_i]\!]_X \neq [\![s_i]\!]_X$. More generally, a *guard* may be a conjunction c of atomic guards c_1 **&** c_2 **&** $\ldots$ **&** c_n, which is true for X_i when each c_j is.

Guarded assignment. This is an instruction denoted $c : a$ of the form **if** c **then** a, where c is a guard and a is a parallel assignment rule. Application of $c : a$ to state X generates the set of updates $\Delta_{c:a}(X) = \bigcup\{\Delta_a(X_i) \ : \ i \in \mathcal{I}, \ [\![c]\!]_{X_i} = \mathsf{true}\}$.

Guarded creation. This instruction c : **new**.a takes the form **if** c **then new** a, where a is a parallel assignment and c, a guard. The assignments are executed on each X_i for which the guard c evaluates to true.

Definition 21 (Parallel Abstract State Program)**.** *A* parallel (abstract state) program *is a finite set $P = \{r_1, \ldots, r_n\}$ of rules as above. To execute P on state X, all rules are executed simultaneously, that is, $\Delta_P(X) = \bigcup_{r_i \in P} \Delta_{r_i}(X)$. If $\Delta_P(X)$ has conflicting updates, then no updates are applied at all.*

For state X, we denote by $P(X)$ the state obtained by application of program P on X. If no rule in P applies, then P is not defined for X.

Note that for each instance of creation, the program chooses new unused indices from $\mathcal{I}$ in some fashion. Since we always treat states and computations as identical if they are the same up to permutation of cells (that is, of indices to function symbols), the specific choice is immaterial.

6.4 Representation Theorem

A parallel program P is a *characteristic program* of algorithm $\mathcal{A}$ if $P(X) = \tau(X)$ for each state X of $\mathcal{A}$. We shall presume for simplicity that $\mathcal{A}$ is over a vocabulary $G \cup F^1$ only and denote it by $G \cup F$. We will also assume that in Postulate VIII we have at most one child born per step ($n = 2$). All proofs can be easily extended to the general case.

By **globalization**, $\Delta_\tau(X) = \bigcup_i \Delta_\tau(X_i)$. So we start with localized states X_i. We prove that the transition of X_i can be described by a rule composed of assignment and creation rules.

By our simplifying assumption, the algorithm has only one local function. So X_i's defined locations are over the vocabulary $G \cup \{f_i\}$. Furthermore, we limit creation to at most one child per transition. Hence, defined locations of $\tau(X_i)$ are over $G \cup \{f_i, f_k\}$ for some $k \in \mathcal{I}$. So we may treat X_i and $\tau(X_i)$ as ordinary states of an ordinary (non-parallel) algorithm over finite vocabulary $G \cup \{f_i, f_k\}$ with critical terms $T_i \cup T_k$.

Let $\delta = h(u_1, \ldots, u_n) \mapsto w$ be an update in $\Delta_\tau(X_i)$. According to [91, Lemma 5] (or [62, Lemma 6.2]), for each value $u_j = 0, \ldots, n$ there exists a term $t^j \in T_i \cup T_k$ such that $\left[\!\left[t^j\right]\!\right]_{X_i} = u_j$. Let α_δ be the ordinary assignment rule $h(t^1, \ldots, t^n) := t^0$. We have $\Delta_{\alpha_\delta}(X_i) = \{\delta\}$, Denote by α_i the assignment obtained as a parallel composition of α_δ for all $\delta \in \Delta_\tau(X_i)$. Obviously, $\Delta_{\alpha_i}(X_i) = \Delta_\tau(X_i)$.

Take a look at $h(t^1, \ldots, t^n) := t^0$, bearing in mind that $\left[\!\left[t^j\right]\!\right]_{X_i} = u_j$ for $j = 0, \ldots, n$. In particular, t^j must be defined (not undef) in X_i. Since the only defined locations of X_i are those of $G \cup \{f_i\}$, we may conclude that t^j are all terms over $G \cup \{f_i\}$, not referring at all to values in the child cell. And since all defined locations of $\tau(X_i)$ are over $G \cup \{f_i, f_k\}$, we may conclude that $h \in G \cup \{f_i, f_k\}$. Accordingly, we partition α_i into two parallel assignment rules: a_i are all those rules with $h \in G \cup \{f_i\}$ and n_i are for rules with $h = f_k$. Obviously, $\alpha_i = a_i \parallel n_i$. We may call the *characteristic assignment* of X_i.

Let $a^\sharp$ be obtained from a_i by replacing f_i with f. Then $a^\sharp$ is an assignment rule over the templates T. From the definition of parallel assignment, we obtain that $\Delta_{a^\sharp}(X_i) = \Delta_{a_i}(X_i)$. Let $n^\sharp$ be obtained from n_i by replacing f_i and f_k with f. From the definitions of parallel creation and of comparing updates for different cells, we obtain that $\Delta_{\mathbf{new}.n^\sharp}(X_i) = \Delta_{n_i}(X_i)$. Define the program $\alpha^\sharp = a^\sharp \parallel \mathbf{new}.n^\sharp$. Then $\Delta_{\alpha^\sharp}(X_i) = \Delta_{a^\sharp}(X_i) \cup \Delta_{\mathbf{new}.n^\sharp}(X_i) = \Delta_{\alpha_i}(X_i)$.

Lemma 22. *Let X_i be a localized state of a parallel algorithm with identity i, and α_i the characteristic assignment for X_i and τ. Then $\alpha^\sharp(X_i) = \alpha_i(X_i) = \tau(X_i)$.*

Proof. That $\alpha^\sharp(X_i)$ is $\tau(X_i)$ follows from the above discussion. That $\alpha_i(X_i)$ is $\tau(X_i)$ follows from [91, Lemma 11]. □

Updates of localized states depend on the values of templates only.

Lemma 23. *Let X_i be a localized state of a parallel algorithm with identity i, and α_i the characteristic assignment for X_i and τ. If Y_i is a localized state with the same identity i and $X_i \equiv_T Y_i$, then $\alpha^\sharp(Y_i) = \alpha_i(Y_i) = \tau(Y_i)$.*

Proof. Since $\alpha^\sharp$ is a rule over T it will contain updates based on the values of T only. Considering that $X_i \equiv_T Y_i$, we will have $\Delta_{\alpha^\sharp}(X_i) = \Delta_{\alpha^\sharp}(Y_i)$. It follows from the previous lemma that $\alpha^\sharp(X_i) = \tau(X_i)$. According to the **localization** postulate, we have $\Delta_\tau(Y_i) = \Delta_\tau(X_i)$, again since $X_i \equiv_T Y_i$. Combining all together, we conclude that $\alpha^\sharp(Y_i) = \tau(Y_i)$, as claimed. □

Every localized state X_i induces an equivalence relation $\sim_{X_i}$ on templates T according to which $s \sim_{X_i} t$ iff $[\![s_i]\!]_{X_i} = [\![t_i]\!]_{X_i}$. We show next that update commands for localization X_i are determined by this relation.

Lemma 24. *Let X_i be an i-state of an algorithm, Y_k an k-state, and α_k the characteristic assignment for Y_k and τ. If $\sim_{X_i} = \sim_{Y_k}$, then $\alpha^\sharp(X_i) = \alpha_k(X_i) = \tau(X_i)$.*

Proof. We may treat X_i and Y_k as ordinary states over finite vocabularies, as we did at the start of this section. We are given that $[\![s_i]\!]_{X_i} = [\![t_i]\!]_{X_i}$ iff $[\![s_k]\!]_{Y_k} = [\![t_k]\!]_{Y_k}$ for any templates $s, t \in T$. By [91, Lemma 13], we get $\alpha_i(X_i) = \tau(X_i)$. By Lemma 22, we may conclude that $\alpha^\sharp(X_i) = \tau(X_i)$. Recall that we consider states to be equal if they are equal up to a permutation of identities. □

We are ready to prove that any parallel algorithm may be described by a parallel program.

Theorem 25 (Representation)**.** *For each parallel algorithm, there exists a characteristic parallel abstract program.*

Proof. For any equivalence relation $\sim$ on templates T, we define the guard $c_\sim$ to be the conjunction of equalities $s = t$ for all $s, t \in T$ such that $s \sim t$, plus the conjunction of disequalities $s \neq t$ for all $s, t \in T$ such that $s \not\sim t$. For each possible relation $\sim$, we choose a localized state X_i of the algorithm with the relation $\sim$ between its instantiated templates T_i (provided there is such a state), and call it $X_\sim$. Then we can define the program $R_\sim$ = **if** $c_\sim$ **then** $\alpha^\sharp_\sim$, where $\alpha_\sim$ is the characteristic assignment for $X_\sim$. Obviously $c_\sim$ evaluates to true on $X_\sim$, and hence $R_\sim(X_\sim) = \alpha^\sharp_\sim(X_\sim)$.

Define P to be the parallel program consisting of rules $R_\sim$ for all possible equivalence relations $\sim$ of T, for which there is at least one state $X_\sim$. Since T is finite, it has only

finitely many distinct equivalence relations, and so program P is finite. We claim that P is a characteristic program of the algorithm, that is, $P(X) = \tau(X)$ for any state X.

Consider some localized state X_i satisfying the relation $\sim_i$ on templates. By Lemma 24, $\alpha_{\sim_i}(X_i) = \tau(X_i)$. Exactly one guard in P applies to X_i and that is $c_\sim$. So $P(X_i) = P_{\sim_i}(X_i) = \alpha^\sharp_{\sim_i}(X_i) = \alpha_{\sim_i}(X_i) = \tau(X_i)$.

Assume finally that X is a general state of the algorithm. By **globalization**, the update of X is a union of updates of all its localizations X_i, that is, $\Delta_\tau(X) = \bigcup_{i\in\mathcal{I}} \Delta_\tau(X_i)$. By the **abstract state** axiom, X_i is also a state. According to **localization**, updates for X_i do not depend on whether X_i is considered as a standalone state or a localization of a general state. So it is enough to show that $\Delta_P(X_i) = \Delta_\tau(X_i)$ for all $i \in \mathcal{I}$, which was just established in the previous paragraph. □

Chapter 7

Extended Computational Thesis

7.1 Introduction

In 1936, Turing [113] invented a theoretical computational model, the Turing machines, and proved that they compute exactly the same functions over the natural numbers (appropriately represented) as do the partial-recursive functions and the lambda calculus. His deep insight was that computation, however complex, can be decomposed into simple atomic steps, consisting of single-step motions and the testing and writing of individual symbols. In 1971, Hartmanis [67] and Cook and Reckhow [29] developed the random-access register machine (RAM) model for the purpose of measuring computational complexity of computer algorithms. This theoretical model is close in spirit to the design of modern (von Neumann architecture) computers and serves as a more realistic measure of (asymptotic) time and space resource usage than do Turing's machines. The question addressed here is to what extent RAMs are in fact the ideal model for measuring algorithmic complexity.

In his handbook survey on computational models, van Emde Boas writes:

> Register-based machines have become the standard machine model for the analysis of concrete algorithms. [117, p. 22]
>
> If it can be shown that reasonable machines simulate each other within polynomial-time bounded overhead, it follows that the particular choice of a model in the definition of feasibility is irrelevant, as long as one remains within the realm of reasonable machine models. [117, p. 4]
>
> I firmly believe that complexity theory, as presently practiced, is based on the following assumption, held to be self evident:

> **Invariance Thesis**: There exists a standard class of machine models, which includes among others all variants of Turing Machines [and] all variants of RAM's.... Machine models in this class simulate each other with Polynomially bounded overhead in time, and constant factor overhead in space. [116, p. 2] (cf. [117, p. 5])

The Church-Turing Thesis [74, Thesis I†] asserts that all effectively computable numeric functions are recursive and, likewise, that they can be computed by a Turing machine, or—more precisely—can be simulated under some representation by a Turing machine. As Kleene [76, p. 493] explains,

> The notion of an "effective calculation procedure" or "algorithm" (for which I believe Church's thesis) involves its being possible to convey a complete description of the effective procedure or algorithm by a finite communication, in advance of performing computations in accordance with it.

This claim has recently been axiomatized and proven [15, 41](see Chapter 2).

The *extended* thesis adds the belief that the overhead in such a Turing machine simulation is only polynomial. One formulation of this extended thesis is as follows:

> The so-called "Extended" Church-Turing Thesis: ... any function naturally to be regarded as efficiently computable is efficiently computable by a Turing machine. (Scott Aaronson [1])

> The Extended Church-Turing Thesis states ... that time on all "reasonable" machine models is related by a polynomial. (Ian Parberry [85])

> The Extended Church-Turing Thesis makes the ... assertion that the Turing machine model is also as efficient as any computing device can be. That is, if a function is computable by some hardware device in time $T(n)$ for input of size n, then it is computable by a Turing machine in time $(T(n))^k$ for some fixed k (dependent on the problem). (Andrew Yao [122])

Indeed, it is widely believed that all effective classical (that is, deterministic, non-parallel, non-analog, non-interactive) models are polynomially-equivalent with regard to the number of steps required to compute. For example, it is well-known that multitape Turing machines (TMs) require quadratic time to simulate RAMs [29] and that single-tape Turing machines require quadratic time to simulate multitape ones [69]. It remains

conceivable, however, that there exists some sort of model that is more sophisticated than RAMs, one that allows for even more time-wise efficient algorithms, yet ought still be considered "reasonable".

We demonstrate that the programs of any classical (sequential, non-interactive) computation model or programming language that satisfies natural postulates of effectiveness (which specialize Gurevich's Sequential Postulates)—regardless of the data structures it employs—can be simulated by a random access machine (RAM) with only constant factor overhead. In essence, the effectiveness postulates assert the following: states can be represented as logical structures; transitions depend on a fixed finite set of terms (those referred to in the algorithm); basic operations can be programmed from constructors; and transitions commute with isomorphisms. Complexity for any domain is measured in terms of constructor operations. It follows that algorithmic lower bounds for the RAM model hold (up to a constant factor determined by the algorithm in question) for any and all effective classical models of computation, whatever its control structures and data structures. This substantiates the Invariance Thesis (a polynomial-time version of the "extended" Church-Turing Thesis), namely that every effective classical algorithm can be polynomially simulated by a Turing machine.

In fact, as will be shown here, RAMs provide *optimal* complexity, regardless of what control structures are available in the programming model and what data structures are employed by the algorithm. Specifically, we show that *any* "effective" computation model (or programming language) can be simulated with only constant slowdown by a (pointer) RAM, measured in terms of basic, constructor operations.

Theorem (Main for Sequential)**.** *(Theorem 32) Any effective classical algorithm using no more than $T(n)$ constructor/destructor operations for inputs of size n can be simulated by a RAM in order $T(n)$ steps, with a word size that may grow up to order* $\log T(n) + \log n$.

We proceed to prove this claim in the following manner:

1. For a generic, datatype-independent notion of algorithm, we adopt the axiomatic characterization of classical *algorithms* over arbitrary domains as was described in Chapter 2.

2. To restrict attention to effective algorithms only—as opposed, say, to conceptual

algorithms like Gaussian elimination over reals—we adopt the formalization of *effective* algorithms over arbitrary domains as was described in Chapter 3.

3. For measuring complexity, we limit operations to basic ones: testing equality of domain elements, application of constructors and destructors, and lookup of stored values (Section 7.2, Definition 26). We would not, for example, normally want to treat multiplication as a unit-cost operation. A *basic* algorithm is one that only employs only these basic operations (Section 3.2, Definition 4).

4. Basic algorithms may be emulated step-by-step with *Extended Storage Modification Machines (EMMs)*, combining results in [62], [5] and [42] (Section 7.4, Lemma 29).

5. Lastly, we show how each step of an EMM can be simulated by a constant number of RAM steps, operating on words of logarithmic size (Section 7.4, Lemma 30).

In [5], it was shown that the emulation can be made precise in that it does not access locations in states that the original algorithm does not.

Having agreed on the right way to measure complexity of effective algorithms over arbitrary data structures (item 3. above), we could have gone on to show directly how to simulate any basic, effective algorithm by means of multidimensional RAMs [93]. Instead, we choose to build (item 4.) on the little-known result of [42], linking a version of EMMs with certain simple abstract state machines, which we will call GASMs. (This work long predates the formalization of effectiveness in [15, 41].) By showing that RAMs simulate EMMs efficiently (which is simpler than showing that they emulate RAMs) and that GASMs emulate any basic implementation of an effective algorithm, the four models (RAM $\geq$ EMM $\geq$ GASM $\geq$ effective algorithm) are chained together and the desired invariance result is obtained in a strong sense—without undue complications.

In [46], the lambda calculus was extended with one-step reduction of primitive operations, and it was shown that any effective computational model can be simulated by this calculus with only constant factor overhead. The catch is—as the authors indicate – that individual steps can themselves be far from basic and quite complex.

Turning to the parallel case, in 1976, Chandra, Kozen, and Stockmeyer [24] proved that alternating polynomial time is equivalent to deterministic polynomial space. In 1977, A. Borodin [18] suggested that this result may be generalized:

> [P]arallel time and space are roughly equivalent within a polynomial factor.

This thesis is usually referred to as the *Parallel Computational Thesis*. In 1978, S. Fortune [47] defined a parallel random access machine (PRAM) and proved that "deterministic parallel RAM's with number processes no more then exponential can accept in polynomial time exactly the sets accepted by Turing machines with polynomially bounded tape." Later, in 1986, Ian Parberry [85] provided an example showing that a Common PRAM with an exponential number of initial processes may compute NPC problems in constant time. He explains that in his opinion this example does not violate the parallel computational thesis but probably this model (PRAM with exponential number of processors at initial state) should not be considered "reasonable":

> [T]he parallel computational thesis does not attempt to say that time on *all* parallel machine models is related; ... it talks only about "reasonable" models.... Thus ... a model is a counterexample to the parallel computational thesis only if it is "reasonable".

Based on the suggestion of Parberry, we do the following: In Chapter 6, we defined a generic parallel algorithm. We restrict that general model to effective ones, similar to the method we used in Chapter 3 for the classical sequential model. That will give us a general effective parallel algorithm. With that in hand, we will prove our main result for parallel models:

Theorem (Main for Parallel)**.** *(Theorem 40) Polynomial time of effective parallel algorithms with no more than an exponential number of cells is equivalent to Turing polynomial space.*

So the only non-Turing possibility for an effective parallel algorithm is when there are more than an exponential number of initial processors.

7.2 Measuring Complexity

The common approach measures (asymptotic) complexity as the (maximum) number of operations relative to input size. As we want to count atomic operations, not arbitrarily complex operations, we should count constructor operations. So we have a choice: to count all the operations executed by an effective algorithm, or to count the transition steps of its corresponding basic algorithm. We take the latter route. To measure the time needed for the execution of a basic algorithm, we use—for the time being—the

"uniform measure" [117, pp. 10–11], under which every transition is counted as a one time unit. Later, we will address the question of what cost to assign to each transition step.

To handle arbitrary data types, the only sensible and honest way is to define the size of a domain element to be the number of basic operations required to build it:

Definition 26 (Size)**.** *The* size *of a domain element is the minimal number of constructor operations required to name that value.*

The size $|n|$ of a unary number n, represented as $s^n(0)$, is $n+1$. The size of n in binary is $\lceil \lg n \rceil$; for example, $|5| = 3$, the length of $0(1(\varepsilon))$, the initial 1(for the string 101) being understood. The size of Turing-machine strings is (one more than) the length of its tape, since string constructors are unary (see the basic Turing-machine implementation in [15]). The size of the tree $C(B(A(), A()), B(A(), A()), B(A(), A()))$ is only 3, because subtrees can be reused, and the whole tree can be specified by

$$C(s, s, s) \text{ WHERE } s = B(r, r), r = A() .$$

An effective algorithm is allowed to access effective oracles (e.g. multiplication) in its initial states, which however are required to be programmable (i.e. algorithmically describable) by a basic algorithm, that is, using constructors and destructors only (usually with a larger vocabulary). In other words, by bootstrapping an effective algorithm, we get a basic one, which is the right one to consider for measuring complexity.

Definition 27 (Complexity)**.** *We measure the complexity of an effective algorithm by the number of basic operations (constructors, destructors, equality) required to perform the computation from initial to final states, relative to the input size.*

In other words, we inline effective sub-algorithms to get a basic one and measure the complexity of the latter.

Consider an effective algorithm `rev` to reverse the top-level elements of a Lisp-like list. The domain consists of all nested lists $\mathcal{L}$; that is, either an empty list $\langle\rangle$, or else a nonempty list of lists: $\langle\langle\rangle\rangle$, $\langle\langle\rangle\langle\rangle\rangle$, $\langle\langle\rangle\langle\rangle\langle\rangle\rangle$, ..., $\langle\langle\langle\rangle\rangle\rangle$, $\langle\langle\langle\rangle\rangle\langle\rangle\rangle$, ..., $\langle\langle\rangle\langle\langle\rangle\langle\langle\rangle\rangle\rangle\rangle$, ... The function `rev`: $\mathcal{L} \to \mathcal{L}$ takes a list $\langle \ell_1 \dots \ell_n \rangle$ and returns $\langle \ell_n \dots \ell_1 \rangle$, with the sublists ℓ_j unchanged. For instance, `rev`$(\langle\langle\rangle\langle\langle\rangle\langle\langle\rangle\rangle\rangle\rangle)$=$(\langle\langle\langle\rangle\langle\langle\rangle\rangle\rangle\langle\rangle\rangle)$.

Now, `rev` could be a built-in operation of the Lisp model of computation, which in one fell swoop reverses any list. Clearly, constant cost for `rev` is not what is intended;

we want to count the number of basic list operations needed to reverse a list of length n. So there is no escape but to take into account how `rev` is implemented internally. Suppose $\texttt{rev}(x)$ is effectively something like this:

$y := x;\ z := \texttt{nil}$

repeat

if $y = \texttt{nil}$

then return z

else [$z := \texttt{cons}(\texttt{car}(y), z);\ y := \texttt{cdr}(y)$]

We want to count the operations executed by this implementation, which is cn for some constant c that is the (maximum) number of (constructor and destructor) operations in a single iteration. Note that any straightforward Turing machine would require many more steps, quadratic in the *size* of the input x, rather than the number of elements at the top level, as in this list-based algorithm. In any RAM implementation, each list is represented by some natural number; what encoding is chosen is immaterial, as long as all operations perform consistently. Regardless of what number is used for the list $\ell = \langle\langle\rangle\langle\langle\rangle\langle\langle\rangle\rangle\rangle\rangle$, $\texttt{car}(\texttt{rev}(\texttt{car}(\texttt{rev}(\ell))))$ should return the number that represents $\langle\langle\rangle\rangle$.

7.3 Machine Models

7.3.1 Random Access Machines

The RAM machine has access to a finite number of registers, and memory locations indexed by non-negative integers; each register or memory location can hold a non-negative integer. For the definition of RAMs, we take the set of instructions suggested by Cook and Reckhow in [29] and use the classification of RAM machines suggested by Boas in [117].

- For *basic RAMs*, the following operations are considered to take "unit time":
  1. $X \leftarrow C$, where X is a register and C is a constant.
  2. $X \leftarrow [Y]$, where $[Y]$ denotes the contents of the memory location indexed by Y.
  3. $[Y] \leftarrow X$.
  4. TRA m if $X > 0$: Transfer control to the m-th line of the program if $X > 0$.

- *Successor RAMs* are an extension of basic RAMs with successor/predecessor operations:

  5. INC X. Increase the value of register X by 1.
  6. DEC X. Decrease the value of register X by 1.

- *Arithmetic RAMs* are the model originally defined by Cook and Reckhow in [29]; they extend basic RAMs with addition and subtraction:

  5. $X \leftarrow Y + Z$.
  6. $X \leftarrow Y - Z$.

- *Multidimensional RAMs* are an extension of the classical ones, allowing for memory organization in multiple dimensions. The instruction set remains the same, but memory cells are accessed using one address per dimension.

Proposition 28 ([93])**.** *Multidimensional arrays may be organized in the memory of a one-dimensional arithmetic RAM in such a way that a program can access an entry indexed by* $[i_1, \ldots, i_\ell]$ *in a constant number of steps, whether or not it is a first-time access (given that a unit instruction can operate over words of size* $O(\log i_1 + \ldots + \log i_\ell)$*).*

The trick is to segment the memory into big chunks in such a way that one can compute squares of indices, hence, cell locations, without need for multiplication; see [93] for details. One can also have more than one array in such a RAM.

7.3.2 PRAMs are Parallel Algorithms

To see how PRAMs meet the requirements we laid out for parallel algorithms, we need to understand what the states would look like from the point of view of our postulates. The domain of the states of a PRAM is the integers (and whatever is isomorphic to the integers). The states are all endowed with the arithmetic capabilities of PRAMs. The global PRAM memory is a global function; the local memories are local; the registers are global or local, as the case may be. The templates are the various registers and expressions appearing in the PRAM program. Forking, however, requires copying all local information to the global area, creating a new cell, and then copying the local information to its proper place.

As we do no allow an effective parallel algorithm to start with initial non-trivial cells, the effective parallel algorithm corresponding to a PRAM would have to first create some input-dependent number of cells and supply them with their local data. The individual cells can also be told what their id is when they are created. Only after setting up such an initial state, from the PRAM's point of view, would one start running the PRAM program proper.

7.3.3 Extended Storage Modification Machines

SMMs [99] manipulate a dynamic pointer structure (while reading an input string and writing to output). Their memory takes the form of a dynamic labeled (multi-) graph. Edges are labeled; nodes are named (not necessarily uniquely) by a path to them from a distinguished *focus* node. A machine may add new nodes to the structure and can redirect edges (perhaps rendering some nodes inaccessible in the process).

Let Λ be a finite alphabet of *direction* labels and X be a finite set of nodes with a distinguished *focus*. For each direction $\delta \in \Lambda$, there is a corresponding function over nodes, such that $\delta(x) = y$ exactly when there is a labelled edge $x \xrightarrow{\delta} y$.

For the convenience, we will denote by $\|W\|$ the end-node of path W.

The set of machine instructions, which may be labeled, is as follows:

- **goto** ℓ—Continue with instruction labeled ℓ.
- **new**W—Create a new node at the end of path W; if W is empty, then the new node becomes the focus; if $W = U\delta$ then the edge labeled δ from the node $\|U\|$ is redirected to a new node; all pointers from this new node are directed to the original $\|W\|$.
- $W := V$—Redirect the last edge of path W to point to the end node of path V.
- **if** $V = W$ **then** P—If paths V and W end at the same node, execute P.
- **if** $V \neq W$ **then** P—If paths V and W end at distinct nodes, execute P.

For example, the instruction **if** $\mathsf{AA} \neq \mathsf{AB}$ **then** $\mathsf{AA} := \mathsf{B}$ has the effect shown in Figure 7.1

We extend the syntax of SMMs with an "inverse" operation, and refer to them as *EMMs*:

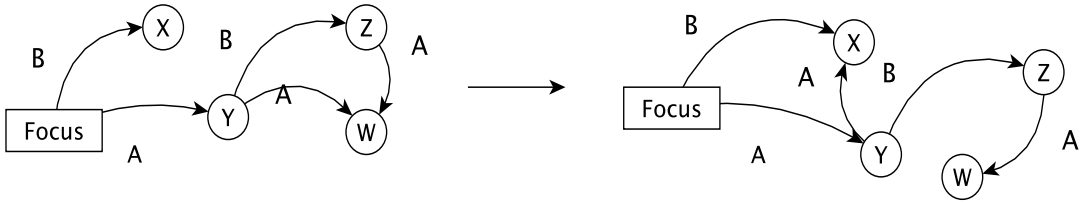


Figure 7.1: SMM: Emulating assignment.

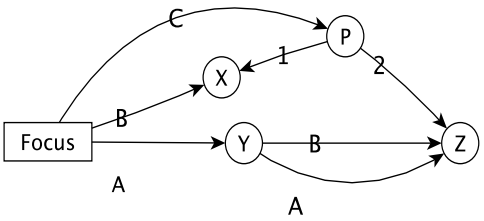


Figure 7.2: Extended SSM: Application of 'Remember' operand

- **remember**$\langle W, V_1, \ldots, V_k, \delta_1, \ldots, \delta_k \rangle$ will remember node $\|W\|$ as the one with edges of type δ_i pointing to nodes $\|V_i\|$, for $i = 1, \ldots, k$. In case of collision with previous applications of **remember**, previously stored values are forgotten.

- **lookup**$\langle V_1, \ldots, V_k, \delta_1, \ldots, \delta_k \rangle$:

  - The machine will set an edge labeled $\langle V_1, \ldots, V_k, \delta_1, \ldots, \delta_k \rangle$ to point from the focus to a node X if X was the last one remembered as the node with edges δ_i pointing to nodes $\|V_i\|$.

  - If there is no appropriate node remembered by the machine, then an edge labeled $\langle V_1, \ldots, V_k, \delta_1, \ldots, \delta_k \rangle$ outgoing from the focus will be removed, if such exists.

For example, performing **remember**$\langle C, B, AA, 1, 2 \rangle$ on the topology the machine will remember the node P as the one with edge of type 1 outgoing to node X and edge of type 2 outgoing to Z. Assume the edges labeled C and 1 have been removed and we request **lookup**$\langle B, AB, 1, 2 \rangle$. The outcome is shown in Figure 7.3.

Note that, despite the fact that P is not reachable from the focus and that its outgoing edges were changed, the machine still remembers it as the one that satisfies the requirement of **lookup**, since it was the last one remembered as such. The set of

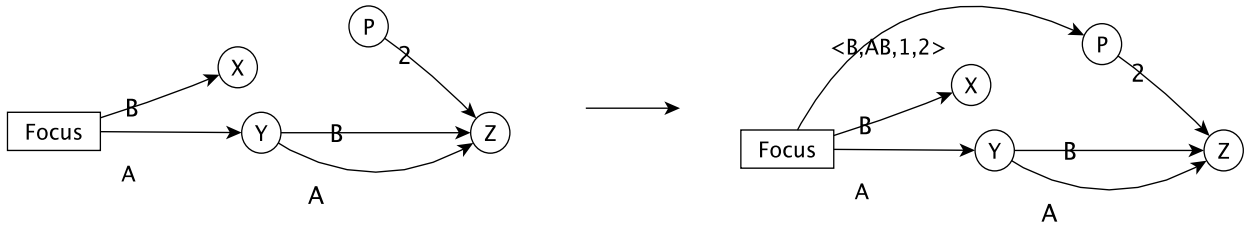


Figure 7.3: Extended SSM: Application of 'Lookup' operand.

edge labels is fixed for any one program, even though these compound labels may be nested.

That all is based on the result from [5], showing that the emulation can be made precise in that it does not access locations in states that the original algorithm does not.

7.3.4 Parallel Random Access Machines (PRAMs)

For the definition of RAMs, we take the set of instructions suggested by Cook and Reckhow in [29] and use the classification of RAM machines suggested by Boas in [117].

For *basic RAMs*, the following operations are considered to take "unit time":

1. $X \leftarrow C$, where X is a register and C is a constant.
2. $X \leftarrow [Y]$, where $[Y]$ denotes the contents of the memory location indexed by Y.
3. $[Y] \leftarrow X$.
4. TRA m if $X > 0$: Transfer control to the m-th line of the program if $X > 0$.
5. READ X. Get next input value.
6. PRINT X. Print to the output tape.

Successor RAMs are an extension of basic RAMs with successor/predecessor operations:

7. INC X. Increase the value of register X by 1.
8. DEC X. Decrease the value of register X by 1.

Arithmetic RAMs are the model originally defined by Cook and Reckhow in [29]; they extend basic RAMs with addition and subtraction:

7. $X \leftarrow Y + Z$.

8. $X \leftarrow Y - Z$.

Multiplication RAMs extend Arithmetic RAMs with multiplication and division:

9. $X \leftarrow Y * Z$.

10. $X \leftarrow Y : Z$.

A *multidimensional RAM* operates a multidimensional memory, rather then one dimensional, as a classical variant. Thus an entry address is defined by a tuple of natural namers.

A parallel RAM (PRAM) consists on several independent sequential processors, each with its own private memory and communicating with one another through a shared (global) memory. In one unit of time, each processor can execute a single RAM operation (and write to one global or local memory location). All processors execute the same RAM program. PRAMs are classified by a type of RAM unit time operations, i.e. in one step of basic PRAM each process can execute one basic RAM instruction. Same for arithmetic and multiplication PRAMs. In addition to this, each process may create a child process, using FORK command. The child process will run the same program as her parent does and will receive from parent the label for the "first command to execute". We use FORK in the way it was pioneered in [47]:

11. FORK *label*. Create a child process which starts execution from *label*.

A *multidimensional PRAM* is a PRAM that has a multidimensional memory, both global and local.

Another important classification is by restriction on shared memory access. In a single step of PRAM, each process can access an entry in shared memory for either reading or writing. And each type of access can be either exclusive (one process access) or common (multiple process access) under some restriction. *The exclusive read/write* restriction prevents reading from/writing to the same global memory cell simultaneously by two distinct processors. We denote these options by *R–read, W–write, E–exclusive, C–common*. So *CREW PRAM* stays for common read exclusive write parallel random access machine. In this type of machine, any process may read any shared memory

entry at any step. But at a single step any entry may be written to by at most one process.

A common write machine should have in its description a restriction for conflict resolution, for a case when multiple processors call for a write to the same global memory cell. Some commonly used methods are: (a) COMMON model:" all processors writing to the same location write the same value" (b) ARBITRARY model: "any process participating in common write may succeed and algorithm should work correctly, regardless of winner" (c) PRIORITY model: "there is a linear order on processors and the one with minimal priority succeeds".

All of the above PRAM models do not differ much in their computational power. PRIORITY PRAM (the strongest model from the above) can be simulated by EREW PRAM (the weakest model from the above) with the same number of processors and with only $O(\log P)$ time overhead, where P is the number of processors [45].

7.4 RAM Simulation of Basic Algorithms

As explained in Section 7.2, we should measure the complexity of effective algorithms in terms of *basic* operations. Thus, we need to show how RAMs simulate basic algorithms. As an intermediary device, we make use of the extension of Schönhage's Storage Modification Machines, EMMs, described in the previous section. We prove that constructor-based algorithms can be simulated by an EMM, which, in turn can be simulated by an arithmetic multidimensional RAM. And all this with negligible overhead. (One could avoid EMMs altogether and show directly how to simulate basic algorithms by multidimensional RAMs, but that would engender the expense of a great deal of technical detail.)

Lemma 29. *Any basic algorithm can be simulated by an EMM with at most constant factor overhead in the number of transitions.*

Proof. A similar claim was proved in [42, Lemma 1] for a different set of algorithms—we'll call them GASMs—and a different extension of SMMs. We prove that GASMs emulate our basic algorithms step-for-step and that our EMMs simulate GASMs with constant-factor overhead in the number of steps.

GASMs satisfy the axiom of algorithmicity, with an added ability to access (**import**) at a single transition a bounded number of fresh (as yet unused) elements.

Without further restrictions on the domain, available oracles and **import** behavior, this class obviously contains also non-effective algorithms, like Euclidian geometry algorithms working over the space of reals or algorithms with access to a Turing halting oracle. Also unrestricted and thus unpredictable behavior of **import** cannot be considered effective. On the other hand, our basic algorithms access domain elements by invoking constructors. So a GASM emulating it will use the same vocabulary as the emulated basic algorithm, and each time a basic algorithm wants to access an element via constructors, a GASM will import a new domain element for that, if that is a first-time access.

It was proved in [42] that any GASM can be simulated for only constant-factor overhead by another GASM whose vocabulary has only nullary (scalar) and unary function symbols plus, optionally, a unique binary symbol used for ordered pairing of elements. The idea behind that is simple: a function of arity n is considered a unary function working over ordered n-tuples. Those n-tuples may be created by $n-1$ applications of pairings. Since the vocabulary of algorithms is finite and depends on algorithm only, the process requires only a bounded number of pairings. They prove that a GASM as above and without pairing can be simulated by a classical SMM. For simulation of pairing, they introduce an extension rule, **create**. An application of **create** V, W, for paths V, W, provides the machine with access to a node representing the ordered pair $\langle \|V\|, \|W\| \rangle$ ($\|V\|$ being the end node of V), in other words, a node with edges labeled `1st` and `2nd` pointing to nodes $\|V\|$ and $\|W\|$, respectively. A machine will use the existing node when possible or else will create a new one (and that despite the fact that the desired node might be inaccessible from the *focus*). To avoid nondeterminism, it is required that any call to pairing be via the **create** rule.

Obviously, **create** may be simulated by our **remember** and **lookup** extensions. Just replace each appearance of **create** by a formal program computing the following:

> Use **lookup**. If nothing is found, use **new** to create the desired node and then use **remember** on it.

Also, any change of edges outgoing that node should be **remember**ed (EMMs with our extension can, in general, **remember** any change it performs). □

Lemma 30. *EMMs can be simulated by multidimensional successor RAMs with at most constant factor overhead in the number of transitions.*

Proof. We give each node $x \in X$ a unique integer identifier $\widehat{x}$. When a new node is created, its identifier will be the successor of the largest previously used integer. In this way, a graph with n nodes uses identifiers $1, \ldots, n$. An identifier for a new node is created using successor.

For each $\delta \in \Lambda$ we define an array, also named δ, and write $\delta[i] = j$ if the contents of the ith entry of array δ is j. (All the arrays can be stored together in one multidimensional array.) To simulate the state of an SMM, these arrays will have the following property:

For all nodes $x, y \in X$, we have $\delta[\widehat{x}] = \widehat{y}$ if and only if $x \xrightarrow{\delta} y$.

A constant focus will contain the identifier of the focus. With this construction, one can find the end point of an edge out of node x labeled δ via a simple query for the value of $\delta[\widehat{x}]$. This can be done in one RAM operation. And an end node of a path of length k can be found in k RAM operations.

Since an SMM program is described finitely, the paths it may query have length bounded by the length of this description. We may conclude from this that, whenever a RAM needs to investigate a path, this path has bounded length and thus the investigation can be performed in a bounded number of RAM operations. It is easy to see that any SMM instruction can be performed using only a bounded number of RAM instructions, corresponding to following and updating edges in the graph.

The extended instruction of EMMs, however, requires the ability to compute the source node of a path. To simulate this, we will have to extend our construction as well. Let $\Lambda = \{\delta_1, \ldots, \delta_k\}$. We define a k-dimensional array A and utilize it to simulate the extension in the following way:

- An application of **remember**$\langle W, V_1, \ldots, V_k, \delta_1, \ldots, \delta_k \rangle$ will be simulated by $A\left[\widehat{\|V_1\|}, \ldots, \widehat{\|V_k\|}\right] := \widehat{\|W\|}$. If an edge labeled δ_i is missing in the description of a command, then the array index will be 0.

- The application of **lookup**$\langle V_1, \ldots, V_k, \delta_1, \ldots, \delta_k \rangle$ will therefore be (remember that an empty path stands for the *focus* node): $\delta\left[\widehat{\|\epsilon\|}\right] := A\left[\widehat{\|V_1\|}, \ldots, \widehat{\|V_k\|}\right]$, with $\delta = \langle V_1, \ldots, V_k, \delta_1, \ldots, \delta_k \rangle$ and with 0 for missing labels.

All the above operations use only assignments and comparisons and may be implemented with only basic RAM commands. A multidimensional RAM with multiple arrays can be easily implemented using one array of large enough dimension. □

Corollary 31. *Any basic algorithm can be simulated by a multidimensional successor RAM with only a constant factor in the number of transitions.*

Proof. To measure the complexity of effective algorithm we "bootstrap" it to a basic one, and then measure the complexity of the latter (see Definition 27). It follows from Lemma 29 that any basic algorithm may be simulated by an EMM, which, by Lemma 30, is doable with a multidimensional successor RAM. □

Everything is in place now to prove our primary result for sequential systems, the Invariance Thesis, which states that every basic algorithm can be simulated by a RAM with the same order of number of steps.

Theorem 32 (Main for Sequential)**.** *Any effective algorithm using no more than $T(n)$ constructor/destructor operations for inputs of size n can be simulated by a (arithmetic) RAM in order $T(n)$ steps, with a word size that may grow to order $\log \max\{T(n), n\}$ bits.*

Proof. It follows from Corollary 31 that any basic algorithm may be simulated by a multidimensional successor RAM, which, in turn, can be simulated by an ordinary arithmetic RAM, by Proposition 28.

Until now, we charged one time unit per transition step, be it a basic algorithm, an EMM, or a RAM. If we desire to count basic operations—constructors and destructors—which is natural for basic algorithms, the overhead will be only a constant factor, since each transition of a basic algorithm may be fully described by a bounded number of invocations of basic operations, where the bound depends on the algorithm only (the maximum number of certain operations) and not on the inputs.

The numbers manipulated by the RAM can grow proportionately to the length of the computation, because there are a bounded number of new domain elements introduced in any one step. So the word length of the RAM is logarithmic in the computation length, plus the length of input if the algorithm is sublinear. □

For a RAM, one might, in fact, wish to charge according to the length of the numbers stored in its arrays [29]. For basic algorithms, counting lookup of values of defined functions and testing equality of domain elements of arbitrary size as atomic operations may also be considered unrealistic. Just as it is common to charge a logarithmic cost for memory access ($\lg x$ to access $f(x)$), it would make sense to place a logarithmic charge $\lg x + \lg y$ on an equality test $x = y$.

Corollary 33. *Basic algorithms (that is, algorithms with basic initial states) and (ordinary) storage modification machines (SMMs) simulate each other to within a constant factor.*

Proof. This follows from the fact that an arithmetic RAM is equivalent time-wise to an SMM, up to a constant factor [99], and that basic algorithms can easily emulate RAMs. □

7.4.0.1 Less Space but More Time

Let $X_1 \rightsquigarrow \cdots \rightsquigarrow X_n$ be a run of a basic algorithm and let a be an element in domain of this run. We say that a is *active* at X_i, for some i, if there is a critical term t whose value over X_i is a; that is *activated* at X_i if there is some $j \leq i$ such that a is active at X_j; and that it is *accessible* at X_i if it can be obtained by a finite sequence of constructor operations on active elements. It was demonstrated in [92] that an element that is accessible at X_i may become inaccessible at X_{i+1}. Inaccessible values should be recycled, much like a Turing machine does not preserve prior values of its tape. The *space usage* of X_i is the minimal number of constructor applications required to construct all active accessible values of X_i. The *space complexity* of a run is the maximal space usage among all states in the run.

For term t, we denote the minimal graph representing it by $G(t)$. Its nodes will each contain a small constant, indicating a vertex label or a pointer, corresponding to an edge in the graph. Note that, since $G(t)$ is minimal, it does not contain repeated factors.

To prevent repeated factors, not just in one term but in the whole state, we merge the individual term graphs (see [86]) into one big graph and call the resulting "jungle" of terms, a *tangle*. The tangle will be used to maintain the constructor-term values of all the critical terms of the algorithm. See [36].

Consider, for example, the natural way to merge terms $t = f(c, c)$ and $s = g(c, c)$, where c is a constant. The resulting directed acyclic graph G has three vertices, labeled f, g, and c. Two edges point from f to c and the other two from g to c. Our two terms may be represented as pointers to the appropriate vertex: $G(t)$ refers to the f vertex and $G(s)$ to g, where we are using the notation $G(t)$ to also refer to the vertex in G that represents the term t. The tangle is shown in Figure 7.4:

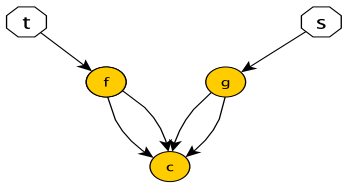


Figure 7.4: An example of tangle representation

On account of the above considerations, tangles are very convenient for distinguishing accessible elements from the inaccessible.

Theorem 34 (Space Invariance Theorem)**.** *Any effective algorithm with time complexity (constructor operations) $T(n)$ and space complexity $S(n)$ can be simulated by an arithmetic RAM in order $nT(n) + T(n)^2$ steps and order $S(n)$ space, with a word size that grows to order $\log S(n)$.*

Proof. In [36], we described how, as an intermediate device, one can simulate basic algorithms using tangles. It is pretty clear how to construct a tangle for a finite set of domain elements. A tangle that simulates state X_i, then, is a tangle for all activated elements of X_i, which we denote by $G(X_i)$. For each critical term t, we keep a pointer, called also t, which points to the value of t in $G(X_i)$. In [36, Thm. 16], it was shown that a basic algorithm with time complexity $T(n)$ can be simulated by a RAM that implements tangles with time complexity $nT(n) + T(n)^2$, using words of size $\log T(n)$.

To obtain the desired result, we need to show that the simulation can be performed with some sort of garbage collection so that each $G(X_i)$ is a tangle of only accessible elements of X_i. Since tangles are acyclic, all we need to do is to maintain a reference count for each node, incrementing it when a new pointer to the node is made and decrementing when a pointer is removed. Whenever the count goes to zero, the node may be added to the free list so that it can be recycled. Only when the free list is empty would a new node be allocated, upping the space usage.

Each of the $T(n)$ constructor operations now comes with a bounded number of additions and subtractions of reference counters by the arithmetic RAM. There is also

the cost of collecting free nodes; indeed, it could be that almost all nodes are recycled in a single step. But amortized, this adds a small constant factor to the time spent by the RAM, since for each creation of a node, of which there are at most $T(n) + n$, there can be at most one freeing up of it. The space overhead is one counter per node, which may double the required space. □

7.5 Effective Parallel Algorithms

We say that a state of parallel algorithm is basic if it is finitely describable. This means that it should be basic in the sense of Definition 4 and have a finite number of processes. Similar to the case in Section 3.2, effective states are an extension of basic states with a finite number of effective oracles. In another words, a state of parallel algorithm is effective if it is effective in a sense of Definition 4 and it should have a finite number of processors.

- We say that a parallel algorithm is basic/effective if its initial state is basic/effective.

- We say that a basic parallel algorithm is in *function-normal form* if its functions (global and local) all but one have arity zero or one, and one constructor of arity two.

Proposition 35. *Any effective parallel algorithm may be emulated by an effective algorithm in function-normal form.*

Proof. This proof for a different case of abstract state machines was first suggested in [43]. The idea is that a function with n arguments is considered as a function with one argument, which is an n-tuple. And an n-tuple is constructed by $n - 1$ applications of pairing. So the emulating algorithm will have a pairing function as one of its constructors. The other functions will have the same names as in original one, but all will be have arities either zero or one. Instead of appealing to $f(u_1, \ldots, u_n)$ it will appeal to $f(\langle u_1, \langle u_2, \langle u_3, \ldots \langle u_{n-1}, u_n \rangle \ldots \rangle \rangle \rangle)$. Since the signature of algorithm is finite, this can be done during the same transition. □

7.6 PRAM Simulation of Basic Parallel Algorithms

Proposition 36. *Any effective parallel algorithm in function-normal form can be simulated by a 3-dimensional Successor Common PRAM with oracle access to some injection $H : \mathbb{N}^3 \to \mathbb{N}$ and with word size big enough for one-step processing of desired H values.*

The overhead in running time is some constant multiplicand, which depends on the simulated algorithm. The number of required processors is equal to the number of cells.

Proof. Let $\mathcal{A}$ be an effective parallel algorithm with global functions $G = \{g_1, \ldots, g_k\}$ and local functions $F = \{f_1, \ldots, f_l\}$. Let X be a state of $\mathcal{A}$. Let D be the domain of X. Let $C \subset G$ be the constructors of D. We choose some order on C, i.e. $C = c_1, \ldots, c_k$. Recall that we identify D with a free-term algebra over C.

1. *Domain Simulation*

   We first define injections $E : D \to \mathbb{N}^3$ and $\mathcal{I}(u) = H(E(u))$ in the following recursive way:

   - $\mathsf{undef} \to (0, 0, 0)$
   - $c_i \to (i, 0, 0)$, when c_i is a constructor of arity zero (i.e. constant);
   - $c_i(u) \to (i, \mathcal{I}(u), 0)$, when c_i is a constructor of arity one, and $u \in D$ is a domain element;
   - $c_i(u_1, u_2) \to (i, \mathcal{I}(u_1), \mathcal{I}(u_2))$, when c_i is the unique constructor element of arity two, and $u_1, u_2 \in D$ are domain elements.

   E and $\mathcal{I}$ are injections since we identified D with a free-term algebra and H is an injection.

2. *Algebra Simulation*

   We are going to describe a multidimensional PRAM state X_H that will simulate X via domain injection $\mathcal{I}$. The PRAM state X_H has the following:

   - a number of processors equal to the number of cells in X,
   - a 3-dimensional shared memory, referred to by G,
   - a 2-dimensional local memory for each processor, referred to by F.

   To each local cell X_i in X we allocate one processor in X_H and we refer to it as p_i. In the shared memory of X_H we will store global values of X. The local memory

of each processor p_i will store local values of cell X_i. The isomorphism of states is defined as follows:

- $G[i,0,0] = \mathcal{I}([\![g_i]\!])$ if g_i is a global constant and zero otherwise;
- $G[i,\mathcal{I}([\![u]\!]),0] = \mathcal{I}([\![g_i(u)]\!])$ if $g_i()$ is a global function and zero otherwise;
- $G[i,\mathcal{I}([\![u]\!]),\mathcal{I}([\![v]\!])] = \mathcal{I}([\![g_i(u,v)]\!])$ for the unique arity-2 constructor $g_i(.,.)$ and zero otherwise;
- $F[i,0] = \mathcal{I}([\![f_{i_k}]\!])$ for some processor p_k if f_i is a local constant and zero otherwise;
- $F[i,\mathcal{I}([\![u]\!])] = \mathcal{I}([\![f_{i_k}(u)]\!])$ if $f_i()$ is a local function and zero otherwise;
- All other entries of shared and local memories are zero.

3. *State Simulation*

   The only information that X_H is missing to simulate X via injection $\mathcal{I}$ is the values of critical terms. Let T be the critical templates of $\mathcal{A}$. For further convenience, we assume that T is closed under the subterm relation (otherwise we take the closure). Then each local cell X_i knows the values of its critical terms T_i. Hence, each processor p_i should keep a pointer for the values of terms in T_i. For this, for each term $t_i \in T_i$ the process p_i will store a constant named t in its local memory. And if $[\![t_i]\!] = u$ at X_i then p_i should store $\mathcal{I}(u)$ as the value of its local constant t. With this information, X_H simulates X via injection $\mathcal{I}$.

4. *Transition Simulation*

   We next show that there exists a program P_H for multidimensional PRAM with oracle access to H such that if $\tau(X) = Y$ then $P_H(X_H) = Y_H$ (where τ is a transition function of algorithm $\mathcal{A}$).

   So let X be a state of $\mathcal{A}$. According to the **globalization** postulate, a transition of X may be viewed as the union of transitions of all local cells X_i of X. According to the **localization** postulate, updates of X_i are the same, whether it is a global state with just one cell or a local cell of a bigger state, i.e. $\tau(X) = \cup_i \tau(X_i)$. Hence it is enough to provide a program P_H such that $P_H(X_{i_H}) = Y_{i_H}$ for any $i \in \mathcal{I}$. In a more general way, it is enough to prove that $P_H(X_H) = Y_H$ for any i-state X.

   Let X be an i-state for some i. Let $Y := \tau(X)$. Let X_H be a PRAM state simulating X, as we described above. Then X_H has only one processor. Let T be critical

templates of $\mathcal{A}$. Let P be a characteristic PASM program of $\mathcal{A}$, as described in Theorem 25. For each transition, P performs a bounded number of basic operations on critical terms: comparisons, assignments and new operations. We should explain how a PRAM may simulate each single basic operation:

- Basic comparison operations ask to compare the values of two critical terms. Since, as we assumed, X_H has those values in special local constants, the unique processor should only compare the values of those two constants. This is done in one single operation, since we assumed that a processor may operate any data entry in one step.

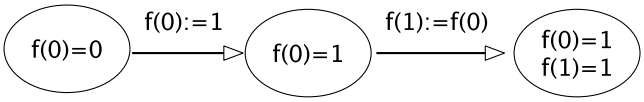
- A basic assignment operation $h(s) := t$ applied on X creates one update $\delta = h(s_X) \mapsto t_X$. We assumed that T is closed under the subterm relation. Hence at X_H we have local values for all terms s and t.

  If h is some global function g_i, an assignment is simulated by a shared memory write command: $G[i, \mathcal{I}([\![s]\!]), 0] \leftarrow \mathcal{I}([\![t]\!])$. If h is some local function f_i, the assignment is then simulated by a local memory write command: $F(i, \mathcal{I}([\![s]\!])) \leftarrow \mathcal{I}([\![t]\!])$. This again can be done in one operation, since we assumed that a processor may operate any data entry in one step.

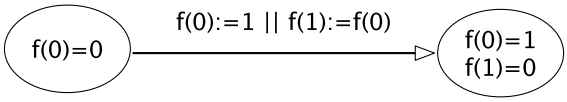
- The **new** operation is simulated by the FORK command of PRAMs. An application of this command returns 0 for a child and the child's process id (pid) for a mother. This provides a way for a process to know that it is a "newborn" one. Some initial information that a mother passes to her child should be created by the mother in shared memory. A mother should wait for the child to copy this information to its local memory and then the mother should clean it. Only after that may the mother move to the next step. Since according to the **motherhood** postulate, a mother may create only a bounded number of data entries for a child, the number of steps required to complete this task is also uniformly bounded. A mother can be programmed to "sleep" (using increment/decrement operations) while its child copies the data. So this action may be simulated in a constant (depending on algorithm) number of PRAM steps.

Note that a PRAM performs a multiple number of assignments in one step. This is not equivalent to sequentially performing the same assignment statements (like

PRAM does). As an example, assume that we have the local value $f(0) = 0$. Consider two assignment statements: $f(0) := 1$ and $f(1) := f(0)$. Sequential application will result in:



whereas simultaneous application will result in:



So to avoid that, before a PRAM starts to construct the updates of X_H, it should make a copy of all critical term values in X_H and use them for reference.

According to the **Algorithmic** postulate, only a bounded number of assignments may be executed in one step. Assume that this bound for our algorithm is m. According to the **Fertility** postulate, only a bounded number of children can be born by one mother in one step. Assume that this bound for our algorithm is n. In addition, according to the **Motherhood** postulate, only a bounded number of data units may be passed from mother to child. Assume that this bound for our algorithm is d.

So to simulate one transition of PSM-Program (and thus of parallel algorithm), a PRAM process should do as described in Algorithm 3.

Sleep pauses are inserted to synchronize the actions of distinct processes.

□

Proposition 37. *Any basic effective parallel algorithm may be simulated by a Multiplication Common PRAM with only constant multiplicand price in running time and with the same number of processors, provided that the PRAM operates on words of logarithmic size.*

Proof. To prove that a Multiplication Common PRAM may simulate basic PASM in function-normal form, according to Proposition 36, we only have to show that we may

Algorithm 3 The parallel RAM simulates one step of a basic parallel ASM Program

4.1. Create a local copy of all critical term values.

4.2. Perform all assignment operations.

- Stay here for exactly m operations (sleep if required).

4.3. Create initial information for a children in shared memory.

- Stay here for exactly $n \cdot d$ operations (sleep if required).

4.4. Call FORK the required number of times and wait for children to update their initial information.

- Stay here for exactly n operations (sleep if required).

4.5. Clean children's information from shared memory.

- Stay here for exactly $n \cdot d$ operations (sleep if required).

4.6. Update critical term values for the next step.

compute some bijection $H : \mathbb{N}^3 \to \mathbb{N}$, preserving the logarithmic size. And this may be computed by the Cantor tuple function:

$$\pi^2(x_1, x_2) = \frac{1}{2}(x_1 + x_2) \cdot (x_1 + x_2 + 1) + x_2$$

$$\pi^n(x_1, \ldots, x^n) = \pi(\pi^{n-1}(x_1, \ldots, x_{n-1}), x_n)$$

Since we are only interested in π^3 we may derive the desired formula and compute it with a bounded number of arithmetic operations - multiplication, addition, division by 2.

Now we only should map the 3-dimensional memory to dimension one. And that can be done again by π^3. □

Proposition 38. *Any Multiplication Common PRAM with time complexity $T(n)$ and with $P(n)$ processors may be simulated be an Arithmetic Common PRAM with $O(\log(n \cdot T(n)) \cdot \log\log^2(n \cdot T(n)) \cdot \log\log\log(n \cdot T(n)))$ time overhead and with $P \cdot \log(n \cdot T(n)) \cdot \log\log(n \cdot T(n)) \cdot \log\log\log(n \cdot T(n))$ processors.*

Proof. It was proved in [100] that multiplication of n-bit numbers can be done by circuits of bounded fan-in with depth $O(\log n)$ and number of agents $O(n \cdot \log n \cdot \log\log n)$ (construction is logspace uniform, i.e. there exists a TM which on input of size n generates in logspace a program executed by each processor). It was proved in [72] that bounded fan-in circuit can be transformed into circuit with bounded both fan-in and fan-out with only constant multiplicand increase a number of gates and in depth.

The latter can be simulated by an Arithmetic EREW PRAM, where gates are simulated by processes and time is equivalent to depth. Obviously, an Arithmetic EREW PRAM may be considered as a special case of an Arithmetic Common PRAM.

Combining the above, an Arithmetic Common PRAM may perform a multiplication of n-bit numbers with an extra $O(n \cdot \log n \cdot \log\log n)$ processes and in $O(\log n)$ time.

In one single step, an Arithmetic PRAM may at most double the number it already has in its memory. So starting with input n, during $T(n)$ steps the maximal value it may generate is $n \cdot 2^{T(n)}$, which can be stored in memory using $\log(n \cdot 2^{T(n)}) = \log n + T(n)$ bits. According to the above, multiplication of numbers with $\log n + T(n)$ bits can be done in $O(\log(\log n + T(n)))$. To do so, one process may require an extra

$$O((\log n + T(n)) \cdot \log(\log n + T(n)) \cdot \log\log(\log n + T(n))$$

processes.

Recall that we simulate a multiplication PRAM. Hence a processor that desires to perform multiplication will have to create its helpers by himself. Thus, it will have to evoke FORK for

$$O((\log n + T(n)) \cdot \log(\log n + T(n)) \cdot \log\log(\log n + T(n))$$

times. And then all those helpers may perform multiplication in $O(\log(\log n + T(n)))$ steps. We may evoke FORK from child processes also, until we got enough processes. Hence, creating n processes will require $\log n$ steps. And hence, creating

$$O((\log n + T(n)) \cdot \log(\log n + T(n)) \cdot \log\log(\log n + T(n))$$

processes may be done in

$$O(\log((\log n + T(n)) \cdot \log(\log n + T(n)) \cdot \log\log(\log n + T(n))) = O(\log(\log n + T(n)))$$

steps. Hence the overall time for one multiplication is still $O(\log(\log n + T(n)))$ steps. The total number of processors used will be:

$$O(P(n) \cdot (\log n + T(n)) \cdot \log(\log n + T(n)) \cdot \log\log(\log n + T(n)))$$

given that $P(n)$ is the number of processors in initial multiplication PRAM. That completes the proof. □

Proposition 39. *Any basic effective algorithm with time complexity $T(n)$ and with $P(n)$ processors can be simulated by an Arithmetic EREW PRAM with time complexity $T(n) \cdot \text{polylog}(n \cdot T(n)) \cdot \text{polylog}(P(n))$ and with $P(n) \cdot \text{polylog}(n \cdot T(n))$ processors.*

Proof. Let $\mathcal{A}$ be a basic effective algorithm with time complexity $T(n)$ and with $P(n)$ processors. It may be simulated by an Arithmetic Common PRAM with time complexity $T(n) \cdot \text{polylog}T(n)$ and with $P(n) \cdot \text{polylog}T(n)$ number of processors, as we proved in Proposition 38. An Arithmetic Common PRAM in its turn may be simulated by EREW PRAM of the same type with only $\log P(n)$ time overhead, were $P(n)$ is the number of processors, as was proved in [45, 118]. Hence, any effective algorithm with time complexity $T(n)$ and with $P(n)$ processors may be simulated by Arithmetic EREW PRAM with time complexity $T(n) \cdot \text{polylog}(n \cdot T(n)) \cdot \text{polylog}P(n)$ and with $P(n) \cdot \text{polylog}(n \cdot T(n))$ processors. □

Theorem 40 (Main for Parallel)**.** *Polynomial time of effective parallel algorithms with number of cells no more then exponential in running time is equivalent to Turing polynomial space.*

Proof. It was proved in [47, Th. 1] that

$$\bigcup_{k=1}^{\infty} T(n)^k\text{-time-Arithmetic-PRAM} = \bigcup_{k=1}^{\infty} T(n)^k\text{-TM-Space}$$

provided that $T(n) \geq \log n$ and the number of processors of the PRAM is no more than exponential in the parallel time. The desired statement thus follows from this and Proposition 39. □

7.7 Discussion

We have shown (Theorem 32) that any algorithm running on any effective classical model of computation can always be simulated by an arithmetic RAM with minimal (linear) overhead and with words of at most logarithmic size, counting constructor operations for effective algorithms. So lower bounds for the RAM model are (up to a

constant factor) lower bounds in an *absolute* sense. We have also seen (Theorem 34) that space complexity may be preserved with only a quadratic increase in time.

It follows that to outperform any RAM, an alternative model must violate one of the postulates. This can be for a number of reasons:

- It is not a discrete-time state-transition system—examples include various analog models of computation, like Shannon's General Purpose Analog Computer (GPAC). See the discussion in [122].
- States cannot be finitely represented as first-order logical structures, all over the same vocabulary—for example models allowing infinitary operations, like the supremum of infinitely many objects.
- The number of updates produced by a single transition is not universally bounded—examples are parallel and distributive models (and probably quantum algorithms, as is widely believed).
- It has a "non-effective" domain—for example, continuous-space algorithms, as in Euclidian geometry or, alternatively, access to non-programmable oracles, like the halting function for Turing machines.

Chapter 8

Generic Cellular Automata

8.1 Introduction

Recent years have seen progress in the understanding of the fundamental notions of computation. We have seen in previous chapters that *abstract state machines (ASMs)* suffice to emulate state-for-state and step-for-step any classical algorithms, as axiomatized by Gurevich [62]; that any algorithm that satisfies an additional effectiveness axiom—regardless of its program constructs and data structures—can be simulated by what we called an *effective ASM*; and that such effective algorithms over arbitrary domains can be efficiently simulated by a random access machine (RAM). In this way, the gap between the informal and formal notions of computation has been reduced, and the classical Church-Turing thesis—that Turing machines entail all manner of effective computation—and its extended version—claiming that "reasonable" effective models have comparable computational complexity—both sit on firmer foundations.

At the same time, von Neumann's cellular model [119] has been enhanced to encompass more flexible forms of computation than were covered by the original model. In particular, the topology of cells can be allowed to change during the evolution of an interconnected device, in what has been called "causal graph dynamics" [2]. Cellular automata have the advantage of better reflecting the laws of physics that a real computing machine must comply with. They respect the "homogeneity" of space-time in that processor cells and memory cells are uniform in nature, in contradistinction with Turing machines, RAMs, or ASMs, whose control are centralized. This cellular approach can help us better understand under what conditions the physical Church-Turing thesis [52], stating that no physically plausible device can compute more functions than a

Turing machine can, might hold [44].

In what follows, we show that any algorithm can be simulated by a dynamic cellular automaton, thus showing that a homogenous physically-plausible model can implement all algorithmic computations. We begin, in the next section, with basic information about cellular automata. It is followed by a description of the simulation and then a brief discussion.

8.2 Background

8.2.1 Cellular Automata

Classical cellular automata are defined as a static tessellate of cells. Initially, each cell is in one of a set of predefined internal states, conventionally identified with colors, of which we will have only finitely many. Sitting somewhere to the side is a clock, and every time it ticks, the colors of the cells change. Each cell looks at the colors of its nearby cells and at its own color and then applies a *transition* rule, specified in advance, to determine the new color it takes on for the next clock tick. Transitions are simple finite-state automata rules. In this model, all cells change at the same time and their transition rules are all the same.

The underlying topology may take different shapes and have different dimensions. The definition of neighborhood may vary from one automaton to another. On a two-dimensional grid, the neighbors may be the four cells in the cardinal directions (called the "von Neumann neighborhood"), or it can include the for corner cells (the "Moore neighborhood"), or perhaps a block or diamond of larger size. In principle, any fixed group of cells of any arbitrary shape can be looked out to determine which transition applies. A *sequential* automaton is the special case when one cell is active and only that cell can perform a transition step. In addition, the transition marks one of the active cell's neighbors (or itself) to be active for the following step.

To model reality better, one should consider the possibility that the connections between cells also evolve over time. For *dynamic cellular automata* [2], cells are organized in a directed graph. Similar to the above classical case, each cell is colored in one of a palette of predefined colors. Edges also have colors, to indicate the type of connection between cells, adding flexibility. Transitions are governed by global clock ticks. In the sequential case one cell is marked active. This cell inspects its neighborhood and

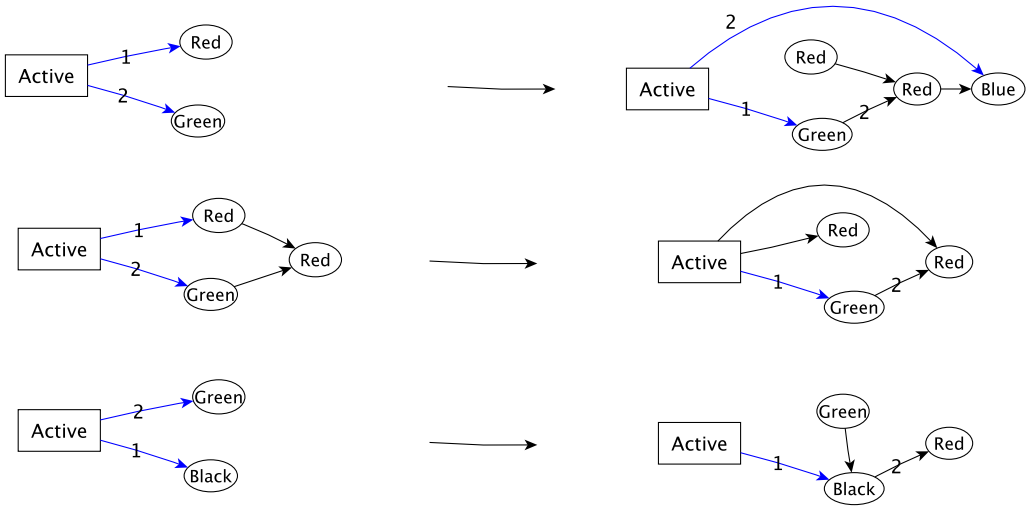


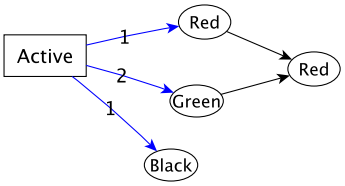
Figure 8.1: Examples of transition rules.

applies a transition rule.

The difference between the static and dynamic cases is that in the static case, the transition is governed by different colorings of the cells in a fixed neighborhood, while in the dynamic case, it is governed by a set of different neighborhood patterns, each with various colored cells connected by colored edges. In both cases, a transition rule defines a transformation of the cells in a detected pattern: in the static case colors change, while in the dynamic case, connections may also change and new cells may be added. With each clock tick, the active cell inspects its neighborhood to detect one of those predefined patterns. Then the transition rule is applied according to the detected pattern. (Cells never die in this model, but they may become disconnected from every other cell.) Examples of such transitions are shown in Figure 8.1.

Note that there might be several transition patterns in the neighborhood of an active cell. For example, given that an active cell detects a pattern of the second type in the example in the figure, it might choose to act according to the first rule instead. If a neighborhood of the active cell contains pattern p, while some subset of its cells also constitute a transition pattern p', we can demand that no transition be applied using p'. We call this restriction *maximality*. (Intuition may be purchased from the following scenario. Assume that your neighbors make a lot of noise from time to time. If at a given time point you have only one noisy neighbor, you might decide to stay put in

peace. But if there are two of them, you would want to call the police. What's worse, if you have three or more rowdy neighbors, you might also need an ambulance. If there is some noise around, a transition might be applied erroneously, as if there were only one noisy neighbor, which is not the natural intent.) So, we want the more specific rules to take precedence over the less constrained ones.[1] In our example, if the second pattern is applicable, then the first one is not applied. All the same, patterns may overlap, so transitions remain non-deterministic. For example, consider the following neighborhood:



In general, these choices can affect the final result, but the simulation we describe has the same outcome regardless.

8.3 Simulating Algorithms with Cellular Automata

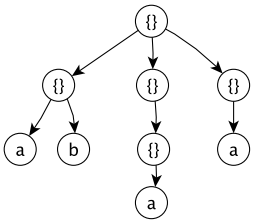
We allow only finitely-describable topologies for cellular automata, and we bound their dynamics, requiring that its transition relation should also be describable by a finite number of patterns.

Our main result is that cellular automata with bounded dynamics can simulate the behavior of any classical algorithm over any unordered domain. We first show how the graph structures of cellular automata can represent the unordered domains of algorithms. Then we show how a transition may simulate manipulations of domain elements.

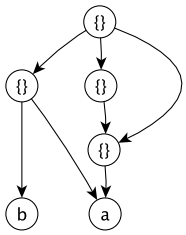
[1] An alternative would be to supply a (partial) order according to which transition rules are tried.

8.3.1 Bounded Dynamics

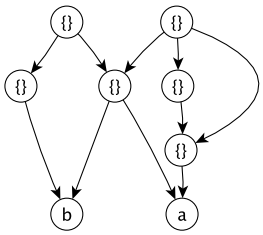
Suppose some domain is constructed over two atoms a and b. The classical tree representation of an element $\{\{\mathsf{a},\mathsf{b}\},\{\{\mathsf{a}\}\},\{\mathsf{a}\}\}$ looks like this:



To avoid obvious reduplication of data, we should use edges pointing to shared locations. This representation is called a *term-graph* [86], and our sample element will look like this:

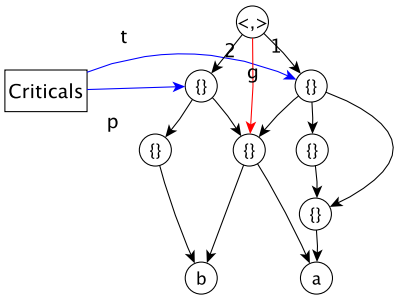


Now, assume that we want to represent two distinct elements $\{\{\mathsf{a},\mathsf{b}\},\{\{\mathsf{a}\}\},\{\mathsf{a}\}\}$ and $\{\{\mathsf{a},\mathsf{b}\},\{\mathsf{b}\}\}$. To avoid reduplication here, we again use pointers to locations shared by both and call the resulting structure a *tangle* [36]. In our example, the tangle will look as follows:



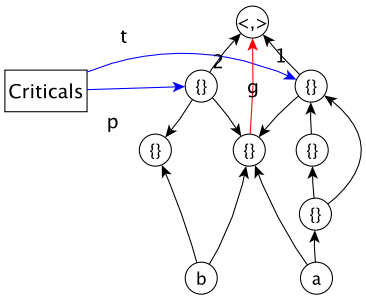
Next, we need to represent the values of functions. We use a slight modification: For each k such that an ASM has a non-constructor function of arity k, we append to the tangle an ordered k-tuple. Assume that our vocabulary has a binary function $g(\cdot,\cdot)$, and assume our ASM has critical terms t and p. Suppose we need to represent state X with values $t = \{\{\mathsf{a},\mathsf{b}\},\{\{\mathsf{a}\}\},\{\mathsf{a}\}\}$, $p = \{\{\mathsf{a},\mathsf{b}\},\{\mathsf{b}\}\}$, and $g(t,p) = \{\mathsf{a},\mathsf{b}\}$. For convenience, we add a focus node called *Criticals*. Edges outgoing from this node point

to the values of critical terms and are labeled appropriately. Our modified tangle will look as follows:



With tangles, we do not have duplicate nodes, that is, no two distinct nodes have the same subtrees, since every domain element is represented by at most one node.

As the last step, we reverse all tangle edges, except for those representing critical terms values, to allow directed access from nodes to parents:



(This step is not necessary, but will have the arrows going in the direction of most of the movements.) Note that both in-degree and out-degree are unbounded.

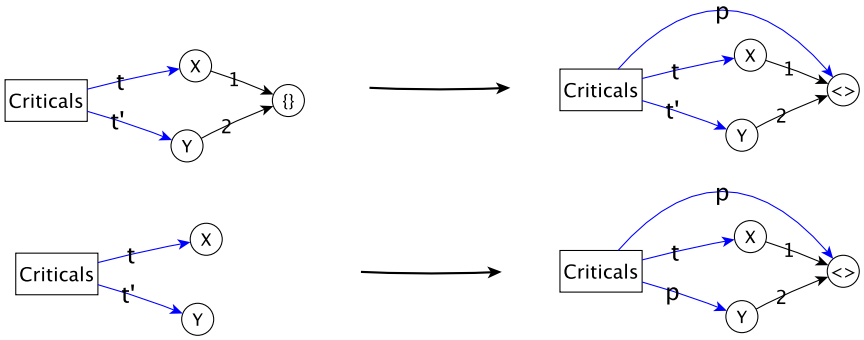
The node labeled Criticals will serve as the active one in the following sequential simulation.

8.3.2 The Simulation

We base the proof of our main result, on the fact that the evolution of any algorithm may be captured by an ASM program. We show that given a domain simulation as above, for each mechanical rule in a program, there is a set of transition rules of a cellular automaton that emulates it. And since each algorithmic transition is described by a finite rule, we will only need finitely many automaton rules to simulate it.

Lemma 41. *Cellular automata simulate the application of pairing* $\langle \cdot, \cdot \rangle$ *in constant time.*

Proof. Assume we want to apply a rule $p := \langle t, t' \rangle$, where t, t', and p are critical terms. The transition rule for the cellular automaton would be as follows:

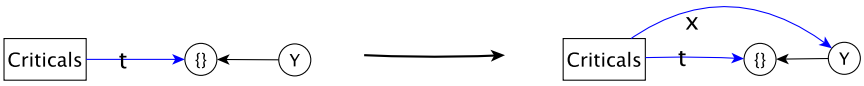


We need the second rule to cover the case when the pair already exists; the first rule is more general and will only fire if the second one is inapplicable.

(The annotations X and Y are not labels; they are used to indicate which nodes on the right of a pattern correspond to which nodes on the left. For convenience, colorless cells like these match a node of any color; skirting formality, this way we need not unnecessarily multiply patterns to cover every possible color combination.) □

Lemma 42. *Cellular automata simulate the application of choice ε in constant time.*

Proof. This operation is used in statements of the form **let** $x = \varepsilon(t)$ **in** A. A straightforward definition of the appropriate transition for a cellular automaton will of necessity be nondeterministic, like the ε operation itself. The pattern chooses the element of t for each of its uses in statement A, like this:



□

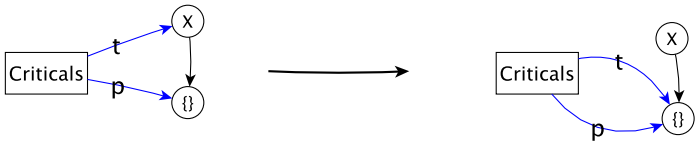
Lemma 43. *Cellular automata simulate the application of conditional tests in constant time.*

Proof. Each transition of an ASM performs a bounded number of actions of two types: Boolean statements and assignments. Since their number is bounded by the algorithm, it is enough for us to describe the simulation of one operation of each type. We have two types of Boolean conditions, inclusions and comparisons:

- Boolean membership $\in$ is used only as a condition. A statement

$$\textbf{if } t \in p \textbf{ then } t := p$$

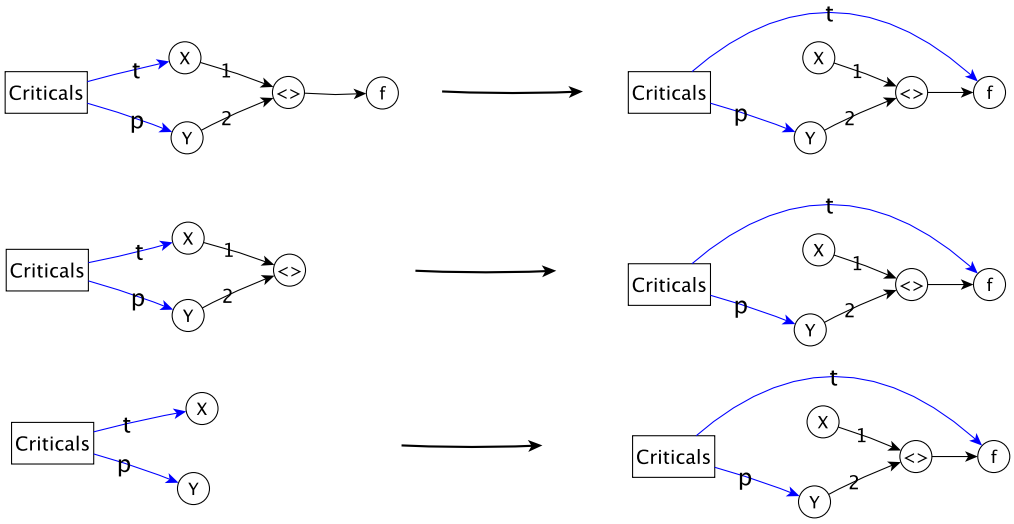
for example, is expressed as follows:



- Boolean comparison is used as a condition. For example, an ASM described by a rule

$$\textbf{if } t \neq p \textbf{ then } t := f(t, p)$$

would be simulated by a cellular automaton with the following transitions to cover all cases (there is a node for $f(t, p)$; there is a node for the pair $\langle t, p \rangle$ but not the value; neither):



□

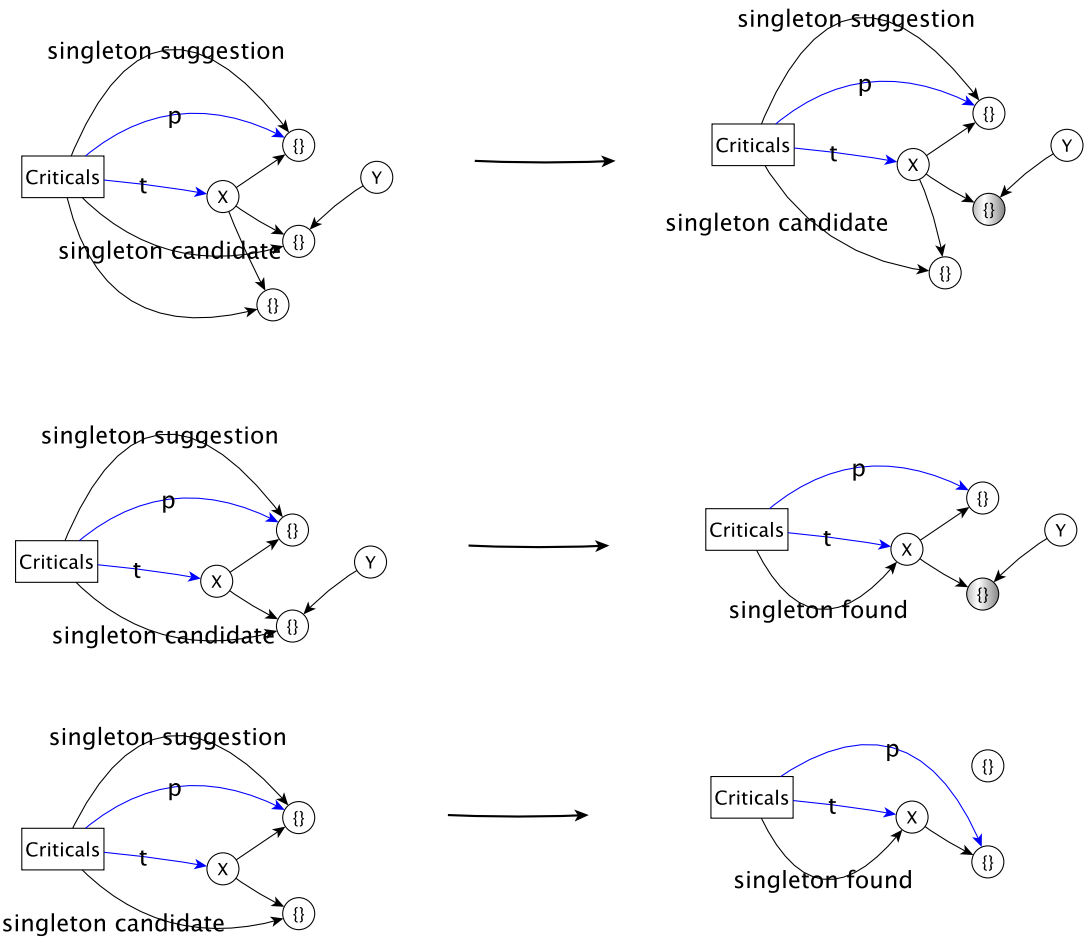
Lemma 44. *Cellular automata simulate the application of singleton formation* $\{\cdot\}$ *in a linear number of steps.*

Proof. Assume that an algorithm applies a rule $p := \{t\}$, where t and p are critical

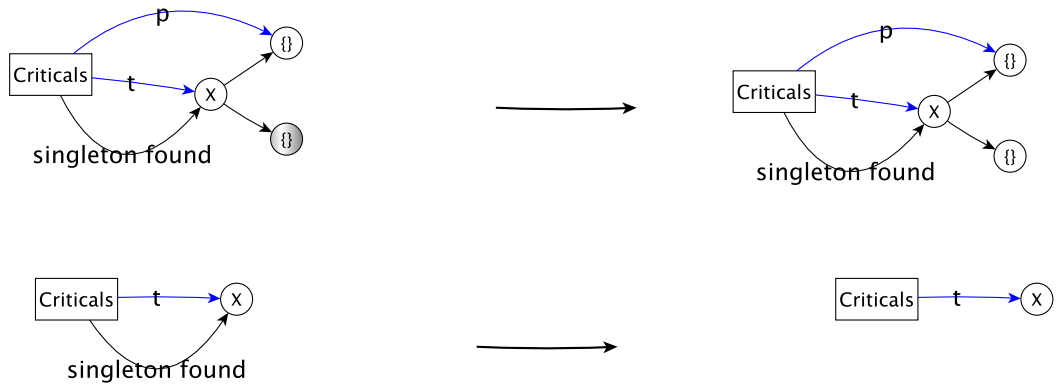
terms. We simulate the singleton operation in three steps. First we create a node for the singleton and mark it singleton suggestion. We also choose another node, if there is one, and mark it singleton candidate:

Then we check if there there already exists a node for that singleton, and if so, we discard the new singleton node created in previous step. To check, we go over all neighbors of X and check each of them in turn. If the requisite singleton node is found, we point to it as the singleton (with a p-marked arrow) and disconnect the newly created node from the tangle. If the tested node is not the desired one, we mark it with a cable that states that the node was tested and move on to the next candidate. When

there are no candidates we mark the newly created node as the desired singleton.



As the last step, we remove the marks from the nodes and then remove the edge singleton found:



As always, we use the rule which forces the most constrained pattern to be applied.

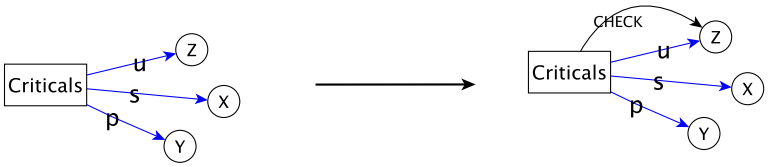
The cost is linear, since we need to check every set of which t is a member to see if it is a singleton. □

Lemma 45. *Cellular automata can simulate applications of the union of two sets with a quadratic number of operations (relative to the number of elements in the sets).*

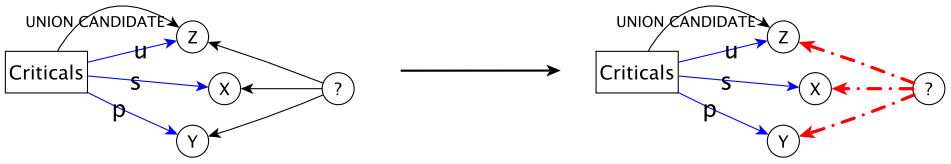
Proof. Suppose we want to simulate the operation $t := s \cup p$, where t, p, s are critical terms, with s pointing to a node indicated by X and p pointing to Y. The simulation will proceed in several stages; the correct order of those steps will be assured by the maximality restriction on transitions. Similar rules should be added to the transition for each possible node coloring. Recall that we want only one instance of each value.

In the beginning, we have to find whether we already have a node representing union of s and p. For this we will go over all accessible nodes from (any) one of the elements that belong in the union. We will show that verifying one node can be done in linear time, so the overall procedure runs in quadratic time.

1. Assume we want to check whether the element whose root is pointed to by u is the union of s and p. We start by creating a special edge to this element. This edge, labeled CHECK, will serve as a lock indicating that we are in the midst of the verification process and will not allow other transitions to get involved in the middle.

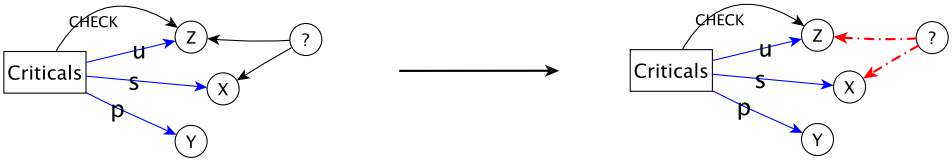


2. Next, we detect the elements appearing in all of u, s and p. Edges from those elements are colored with a special color:



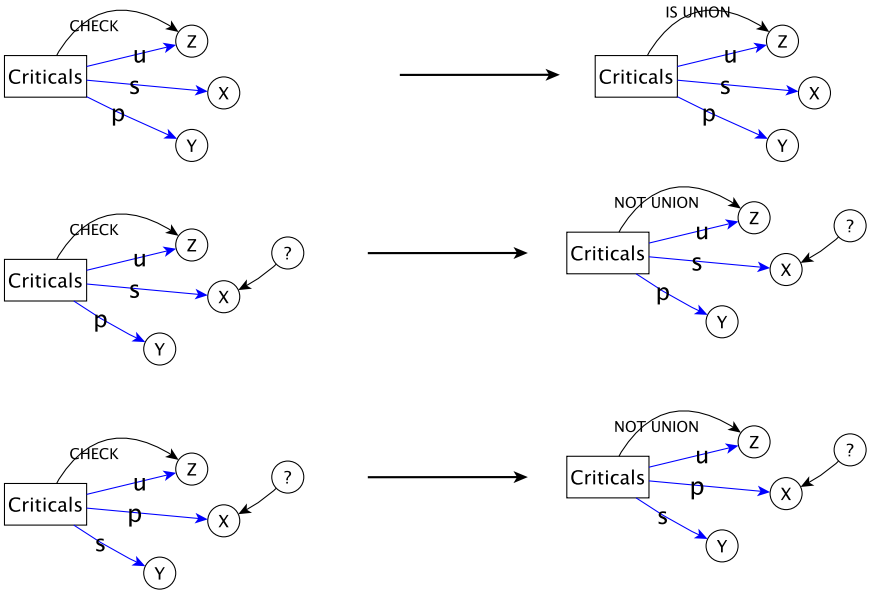
The same is done (in parallel) with t and p.

3. Next, we detect common elements of u and s but not in p. Pointers from detected elements are again marked with the special color:

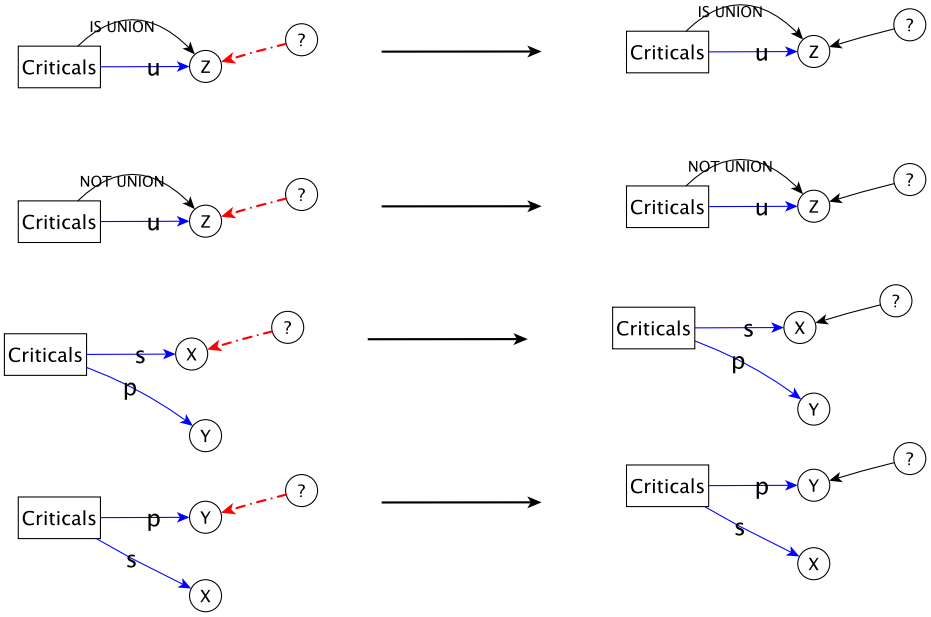


The same is done with u and p.

4. If s, p, and u are all empty, then u is indeed the union of s and p. Mark it as such. Otherwise, u is not the union node:



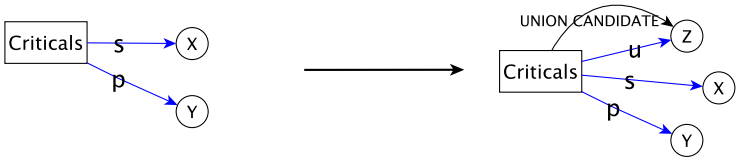
5. Once the status of u is clear, we remove the marks from edges:



Each element identified to not be the union is marked with a special color for the duration of the search so as not to re-check it, similar to the singleton case.

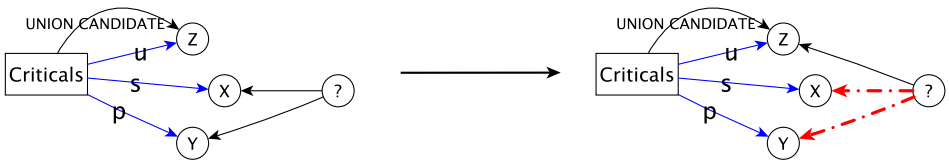
Once all the possible elements have been checked, and no union found, we are ready to create the union.

1. First, we create a new node which will eventually hold the union:

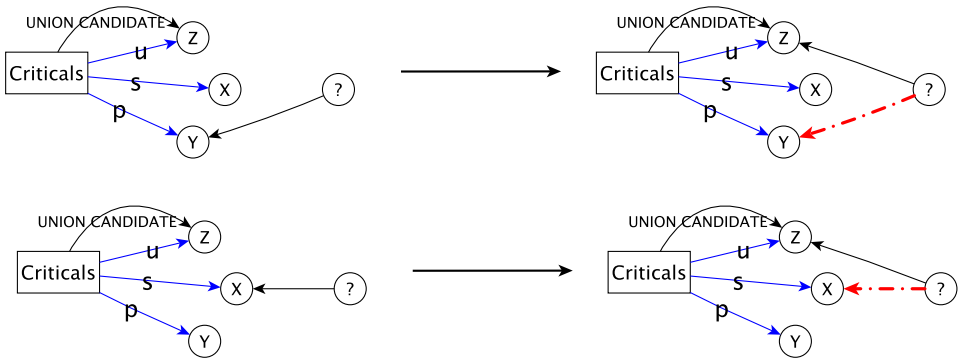


A special marked edge tells the automaton that it is the process of creating a union.

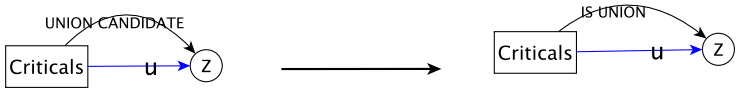
2. We start with copying to u the elements that are common to s and p, and mark the edges:



3. In a similar manner, we copy elements that are present in one set only:

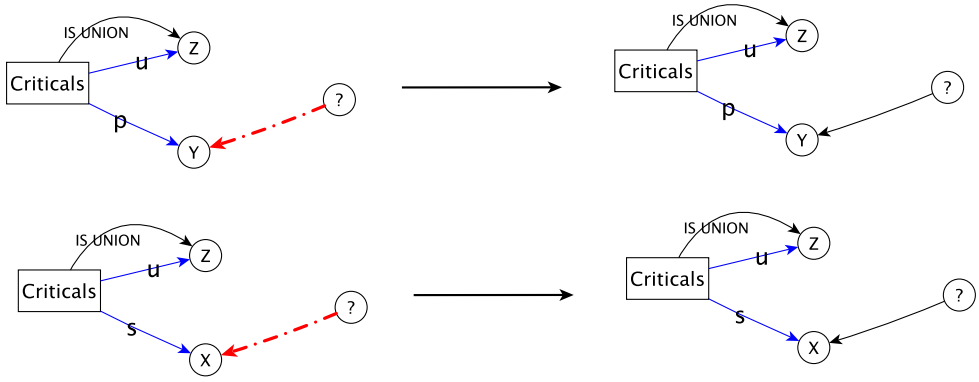


4. Once all edges are marked, we know that the desired node is created and we mark it appropriately:

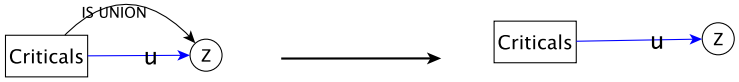


This rule applies only when no element transfers remain.

5. Once the IS UNION mark appears, all that remains is to clean the marks left en route:



6. As soon as the union is ready and its neighborhood is clean, we may remove the lock:



Note again that the maximality restriction on transitions ensures that all the above steps are applied in the prescribed order. □

We know that every classical algorithm is emulated step-by-step, state-by-state by an ASM consisting of a fixed number of comparisons and assignments [62]. That fact, along with the previous lemmata, is what is needed to achieve the main goal of this chapter:

Theorem 46 (Main)**.** *Cellular automata with bounded dynamics (i.e. all the nodes in a pattern are within a bounded distance of the focus) and without loops (there are no directed cycles within patterns) can simulate the performance of any classical algorithm over an unordered domain with quadratic multiplicand overhead.*

Proof. We have to ensure that, once the automaton starts to simulate the singleton or union operation, it cannot be interrupted by the application of other transition rules. Otherwise, foreign steps could affect the elements of the sets involved in these set operations. This problem can be precluded, for instance, by changing the color of the Criticals node during the simulation of those operations.

Each step of the original algorithm can only create a bounded number of new sets. Hence the size of the sets involved in any union operation is bounded by the size of the sets in the initial state plus some multiple of the algorithm's steps so far. So the overall overhead caused by unions is quadratic. □

Chapter 9

Continuous Time

9.1 Introduction

We seek to gain an understanding of the fundamentals of analog systems, that is, systems that operate in continuous (real) time and with real values. Several different approaches have been taken in the pursuit of continuous-time models of computation. One is inspired by continuous-time analog machines, and has its roots in models of natural or artificial analog machinery. An alternate approach, one that can be referred to as inspired by continuous-time system theories, is broader in scope, and derives from research in systems theory done from a computational perspective. Hybrid systems and automata theory, for example, are two such sources of inspiration. See the survey in [19].

At the outset, continuous-time computation theory was mainly concerned with analog machines. Determining which systems should actually be considered to be algorithmic in nature is an intriguing question and relates to philosophical discussions about what constitutes a programmable machine. All the same, there are a number of early examples of actual analog devices that are generally accepted to be programmable. These include Pascal's 1642 *Pascaline* [30], Hermann's 1814 *Planimeter*, Bush's landmark 1931 *Differential Analyzer* [22], as well as Bill Phillips' 1949 water-run *Financephalograph* [121]. Continuous-time computational models also include neural networks and systems that can be built using electronic analog devices. Such systems begin in some initial state and evolve over time in response to input signals. Results are read off from the evolving state and/or from a terminal state.

Another line of development of continuous-time models was motivated by hybrid

systems, particularly by questions related to the hardness of their verification and control. In this case, the models are not seen as models of necessarily analog machines, but, rather, as abstractions of systems about which one would like to establish some properties or derive verification algorithms.

Our goal is to capture all such models within one uniform notion of computation and of algorithm. The most interesting case is the hybrid one, where the system's dynamics change in response to changing conditions, so there are discrete transitions as well as continuous ones. To that end, we adopt and adapt some of the ideas embodied in Gurevich's abstract-state machine formalism for discrete algorithms [61], which we have been using throughout this dissertation.

Capturing the notion of algorithmic computation for analog systems is a first step towards a better understanding of computability theory for continuous-time systems. Even this first step is a non-trivial task. Some work in this direction has been done for simple signals. See, for example, [27, 28] for an approach within the abstract-state machine framework. An interesting approach to specifying some continuous-time evolutions, based on abstract state machines and using infinitesimals, is [98]. However, a comprehensive framework, capturing general analog systems seems to be wanting. See [19] for a discussion of the diverse analog computability theories.

In this chapter, we adapt and extend ideas from work on ASMs to the analog case, that is to say, from notions of algorithms for digital models to analogous notions for *analog systems*. We go beyond the easier issue of "continuous space", that is, discrete-time models or algorithms with real-valued operations, since these have already been made to fit comfortably within the ASM framework, for which, see [5]. Indeed, algorithms for discrete-time analog models, like algorithms for the Blum-Shub-Smale model of computation [11], can be covered in this setting. The geometric constructions in [92] are simple (loop-free) examples of continuous-space algorithms.

In the next section, we introduce dynamical transition systems, defining signals and transition systems. In Section 9.3, we introduce abstract dynamical systems. Next, in Section 9.4, we define what an algorithmic dynamical system is. Then, in Section 9.5, we define analog programs and provide some examples, followed by a brief conclusion.

9.2 Dynamical Transition Systems

Analog systems may be thought of as "states" that evolve over "time". The systems we deal with receive inputs, called "signals", but do not otherwise interact with their environment.

9.2.1 Signals

Typically, a signal is a function from an interval of time to a "domain" value, or to a tuple of atomic domain values. For simplicity, we will presume that signals are indexed by real-valued time $\mathbb{T} = \mathbb{R}$, are defined only for a finite initial (open or closed) segment of $\mathbb{T}$, and take values in some domain D. Usually, the domain is more complicated than simple real numbers; it could be something like a tuple of infinitesimal signals. Every signal $u : \mathbb{T} \rightharpoonup D$ has a *length*, denoted $|u|$, such that $u(j)$ is undefined beyond $|u|$. To be more precise, the length of signals that are defined on any of the intervals $(0, \ell), [0, \ell), (0, \ell], [0, \ell]$ is ℓ. In particular, the length of the (always undefined) *empty* signal, ε, is 0, as is the length of any point signal, defined only at moment 0.

The *concatenation* of signals is denoted by juxtaposition, and is defined as expected, except that concatenation of a right-closed signal with a left-closed one is only defined if they agree on the signal value at those closed ends, and concatenation is not defined if they are both open at the point of concatenation. The empty signal ε is a neutral element of the concatenation operation.

Let $\mathcal{U}$ be the set of signals for some particular domain D. The *prefix* relation on signals, $u \leq v$, holds if there is a $w \in \mathcal{U}$ such that $v = u\,w$. As usual, we write $u < v$ for *proper* prefixes ($u \leq v$ but $u \neq v$). It follows that $\varepsilon \leq u \leq uw$ for all signals $u, w \in \mathcal{U}$. And, $u \leq v$ implies $|u| \leq |v|$, for all u, v.

9.2.2 Transition Systems

Definition 47 (Transition System)**.** *A* transition system $\langle \mathcal{S}, \mathcal{S}_0, \mathcal{U}, \mathcal{T} \rangle$ *consists of the following:*

- *A nonempty set (or class)* $\mathcal{S}$ *of* states *with a nonempty subset (or subclass)* $\mathcal{S}_0 \subseteq \mathcal{S}$ *of* initial *states.*
- *A set* $\mathcal{U}$ *of input signals over some domain* D*.*

- *A $\mathcal{U}$-indexed family $\mathcal{T} = \{\tau_u\}_{u\in\mathcal{U}}$ of* state transformations $\tau_u : \mathcal{S} \to \mathcal{S}$.

Initial states might, for example, differ in the values of parameters, such as initial values.

It will be convenient to abbreviate $\tau_u(X)$ as just X_u, the state of the system after receiving the signal u, having started in state X. We will also use $X_{\widetilde{u}}$ as an abbreviation for the *trajectory* $\{X_v\}_{v<u}$, describing the past evolution of the state.

For simplicity, we are assuming that the system is deterministic. Note that the classical ASM framework for digital algorithms, though initially defined for deterministic systems, has been extended to nondeterministic transitions in [53, 65].

Should one want to model the possibility of *terminal* states, then the transformations would be partial functions $\tau_u : \mathcal{S} \rightharpoonup \mathcal{S}$. We gloss over this distinction in what follows.

Definition 48 (Dynamical System)**.** *A* dynamical system $\langle \mathcal{S}, \mathcal{S}_0, \mathcal{U}, \mathcal{T}\rangle$ *is a transition system, where the transformations satisfy*

$$\tau_{uv} \quad = \quad \tau_v \circ \tau_u,$$

for all $u, v \in \mathcal{U}$, and where τ_ε is the identity function on states.

This implies that $X_{uv} = (X_u)_v$.

It follows from this definition that $\tau_{(uv)w} = \tau_{u(vw)}$, since composition is associative. It also follows that instantaneous transitions are idempotent. That is, $\tau_a \circ \tau_a = \tau_a$, for point signal a, because then $aa = a$.

9.3 Abstract Dynamical Systems

9.3.1 Abstract States

A vocabulary $\mathcal{F}$ is a finite collection of fixed-arity function symbols, some of which may be tagged *relational*. A term whose outermost function name is relational is termed *Boolean*. The following definition extends the notion of algorithm (Definition 1) from discrete systems, with which we have been working until now, to the analog case:

Definition 49 (Abstract Transition System)**.** *An* abstract transition system *is a dynamical transition system whose states $\mathcal{S}$ are (first-order) structures over some finite vocabulary $\mathcal{F}$, such that the following hold:*

1. *States are closed under isomorphism, so if $X \in \mathcal{S}$ is a state of the system, then any structure Y isomorphic to X is also a state in $\mathcal{S}$, and Y is an initial state if X is.*

2. *Input signals are closed under isomorphism, so if $u \in \mathcal{U}$ is a signal of the system, then any signal v isomorphic to u (that is, maps to isomorphic values) is also a signal in $\mathcal{U}$.*

3. *Transformations preserve the domain (base set); that is,* $\mathrm{Dom}X_u = \mathrm{Dom}X$ *for every state $X \in \mathcal{S}$ and signal $u \in \mathcal{U}$.*

4. *Transformations respect isomorphisms, so, if $X \cong_\zeta Y$ is an isomorphism of states $X, Y \in \mathcal{S}$, and $u \cong_\zeta v$ is the corresponding isomorphism of input signals $u, v \in \mathcal{U}$, then $X_u \cong_\zeta Y_v$.*

In particular, system evolution is *causal* ("retrospective"): a state at any given moment is completely determined by past history and the current input signal. This is analogous to the postulates for discrete algorithms except that subsequent states X_u depend on the whole signal u, not just the prior state X and current input.

To keep matters simple, we are assuming (unrealistically) that all operations are total. Instead, we simply model partiality by including some *undefined* element $\perp$ in domains. See, however, the development in [5].

9.3.1.1 Vocabularies.

We will assume that the vocabularies of all states include the Boolean truth constants, the standard Boolean operations, equality, and function composition, and that these are always given their standard interpretations. We treat predicates as truth-valued functions, so states may be viewed as algebras.

There are idealized models of computation with reals, such as the BSS model [11], for which true equality of reals is available in all states. On the other hand, there are also models of computable reals, for which "numbers" are functions that approximate the idealized number to any desired degree of accuracy, and in which only partial equality is available. See [5] for how to extend the abstract-state-machine framework to deal faithfully with such cases.

9.3.2 Updates of States

As in the classical case, we need to capture the changes to a state that are engendered by a system. For a given abstract transition system, define its *update function* Δ as follows:

$$\Delta(X) = \lambda u.\ X_u \setminus X$$

We write $\Delta_u(X)$ for $\Delta(X)(u)$. The trajectory of a system may be recovered from its update function, as follows:

$$X_u \quad = \quad (X \setminus \nabla_u(X)) \cup \Delta_u(X)$$

where

$$\nabla_u(X) := \{\ell \mapsto [\![\ell]\!]_X : \ell \mapsto b \in \Delta_u(X) \text{ for some } b\}$$

are the location-value pairs in X that are updated by Δ_u.

9.4 Algorithmic Dynamic Systems

A *template* is a term over the algorithm's vocabulary containing a variable. Let $t(x)$ be a template, u a domain element, and X a state. The variable x formalizes the idea of an incoming port, which receives a signal u.

We denote by $[\![t_u]\!]_X$ the value of $t(u)$ on X (i.e. we substitute u for x and compute its value in X). We say that states X and Y *agree* on t_u if $[\![t_u]\!]_X = [\![t_u]\!]_Y$. Let T be a set of critical templates, all with the same variable x and u, a domain element. Let T_u be $\{s_u \ : \ s \in T\}$. We say that states X and Y *agree* on T_u if $[\![s_u]\!]_X = [\![s_u]\!]_Y$ for all $s \in T$. This will be abbreviated $X =_{T_u} Y$. We also say that states X and Y are *similar*, with respect to T_u if, for all templates $s, t \in T$, we have $[\![s_u]\!]_X = [\![t_u]\!]_X$ iff $[\![s_u]\!]_Y = [\![t_u]\!]_Y$. This will be abbreviated $X \sim_{T_u} Y$.

9.4.1 Algorithmicity

The current state, "modulo" its critical terms, unambiguously determines future states.

Definition 50 (Algorithmic Transitions)**.** *An abstract transition system with states $\mathcal{S}$ over vocabulary $\mathcal{F}$ is* algorithmic *if there is a fixed* finite *set T of* critical *templates over $\mathcal{F}$, such that $\Delta_u(X) = \Delta_u(Y)$ for any two of its states $X, Y \in \mathcal{S}$ and signal $u \in \mathcal{U}$,*

whenever X and Y agree on T. In symbols:

$$X =_{T_u} Y \quad \Rightarrow \quad \Delta_u(X) = \Delta_u(Y)\,.$$

This implies

$$X_{\widetilde{u}} =_{T_u} Y_{\widetilde{u}} \quad \Rightarrow \quad \Delta_u(X) = \Delta_u(Y)\,.$$

Furthermore, similarity should be preserved:

$$X_{\widetilde{u}} \sim_{T_u} Y_{\widetilde{v}} \quad \Rightarrow \quad X_{ua} \sim_{T_u} Y_{va}\,,$$

where $a \in \mathcal{U}$ is any point signal ($|a| = 0$).

Following the reasoning in [62, Lemma 6.2], every new value assigned by $\Delta_u(X)$ to a location in state X is the value of some critical template. That is, if $\ell \mapsto b \in \Delta_u(X)$, then $b = [\![t_u]\!]_X$ for some critical $t \in T$.

Proposition 51. *Every new value assigned by $\Delta_u(X)$ to a location in state X is the value of some critical term. That is, if $\ell \mapsto b \in \Delta_u(X)$, then $b = [\![t_u]\!]_X$ for some critical $t \in T$.*

Proof. By contradiction, assume that some b is not critical. Let Y be the structure isomorphic to X that is obtained from X by replacing b with a fresh element b'. By the abstract-state postulate, Y is a state. Check that $[\![t_u]\!]_Y = [\![t_u]\!]_X$ for every critical template t. By the choice of T, $\Delta_u(Y)$ equals $\Delta_u(X)$ and therefore contains b in some update. But b does not occur in Y. By (the inalterable-base-set part of) the abstract-state postulate, b does not occur in Y_u either. Hence it cannot occur in $\Delta_u(Y) = Y_u \setminus Y$. This gives the desired contradiction. □

Agreeability of states is preserved by algorithmic transitions:

Lemma 52. *For an algorithmic transition system with critical templates T, it is the case that*

$$X =_{T_u} Y \quad \Rightarrow \quad X_u =_{T_u} Y_u$$

for any states $X, Y \in \mathcal{S}$ and input signal $u \in \mathcal{U}$.

9.4.2 Flows and Jumps

A "jump" in a trajectory is a change in the dynamics of the system, in contrast with "flows", during which the dynamics are fixed. Formally, a jump corresponds to a change in the equivalences between critical terms, whereas, when the trajectory "flows", equivalences between critical terms are kept invariant. Accordingly, we will say that a trajectory $X_{\widetilde{u}}$ *flows* if all intermediate states X_w and X_v ($\epsilon < w < v < u$) are similar. It *jumps* at its end if there is no prefix $w < u$ such that all intermediate X_v, $w < v < u$, are similar to X_u. It *jumps* at its beginning if there is no prefix $w \leq u$ such that all intermediate X_v, $\epsilon < v < w$, are similar to X.

9.4.3 Analgorithms

Putting everything together, we have arrived at the following.

Definition 53 (Analog Algorithm)**.** *An* analog algorithm *(or "analgorithm") is an algorithmic (abstract) transition system, such that no trajectory has more than a finite number of (prefixes that end in) jumps.*

In other words, an analog algorithm is a signal-indexed deterministic state-transition system (Definitions 47 and 48), whose states are algebras that respect isomorphisms (Definition 49), whose transitions are governed by the values of a fixed finite set of terms (Definition 50), and whose trajectories do not change dynamics infinitely often (Definition 53).

9.4.4 Properties

System evolution is *causal* ("retrospective"): a state at any given moment is completely determined by past history and the current input signal.

Theorem 54. *For any analog algorithm, the trajectory can be recovered from the immediate past (or updates from the past). In other words, X_u, for right-closed signal u, can be obtained (up to isomorphism) as a function of $X_{\widetilde{u}}$ (that is, the X_v, for $v < u$) plus the final input u_*.*

In fact, X_u depends on arbitrarily small segments $X_{u(t,|u|)}$, $t < |u|$, of past history.

Proof. This is a direct consequence of Definition 49. □

9.4.5 Further Considerations

It might also make sense to disallow the value given to a location ℓ at some time t to depend on infinitely many prior changes. For example, one would not want the value of $f(t)$ to be set at every moment t to $2f(t/2)$. Rather, the value of every location ℓ at moment t should be determined by values provided by the signal at time t and by values of locations in the state that are "stable" at t. By *stable*, we mean that there is a non-empty interval of time up to t in which its value is constant. Furthermore, this temporal dependency of locations should be well-founded.

It may happen that the system of equations that controls transitions has a critical non-unique solution for the given initial conditions. For example, the equation $y'(x)^2 = 4y(x)$, restricted to the initial condition $y(0) = 0$, has two distinct solutions, namely, $y \equiv 0$ and $y = x^2$. In this case, we would want to add some continuity constraint. We would want to require that a choice of the solution made in the initial state is not changed for the whole trajectory governed by that equation.

9.5 Programs

9.5.1 Definition

Definition 55 (Analog Program)**.** *An* analog program P *over a vocabulary* $\mathcal{F}$ *is a finite text, taking one of the following forms:*

- *A* constraint *statement* $v_1, \ldots, v_n$ ***s.t.*** C*, where* C *is a Boolean condition over* $\mathcal{F}$ *and the* v_i *are terms over* $\mathcal{F}$ *(usually subterms of* C*) whose values may change in connection with execution of this statement.*
- *A* parallel *statement* $[P_1 \parallel \cdots \parallel P_n]$ *(*$n \geq 0$*), where each of the* P_i *is an analog program over* $\mathcal{F}$*. (If* $n = 0$*, this is "do nothing" or "skip".)*
- *A* conditional *statement* **if** C **then** P*, where* C *is a Boolean condition over* $\mathcal{F}$*, and* P *is an ASM program over* $\mathcal{F}$*.*

We can use an assignment statement $f(s_1, \ldots, s_n) := t$ as an abbreviation for $f(s_1, \ldots, s_n)$ **s.t.** $f(s_1, \ldots, s_n) = t$. But bear in mind that the result is instantaneous, so that $x := 2x$ is tantamount to $x := 0$, regardless of the prior value of x. Similarly, $x := x + 1$ is only possible if the domain includes an "infinite" value ∞ for which $\infty = \infty + 1$.

9.5.2 Semantics

In the simple case, where the changes in state at time t depend only on the current signal u and state X, we can envision the following sequence of events:

1. All non-stable locations in X (see Section 9.4.5) have undefined values.

2. The signal sets the value of location $\imath$, yielding X'.

3. Critical terms are evaluated in X'. (Only relevant terms need be evaluated, per [5].) This may involve looking up the values of pre-defined "static" operations in the state, like multiplication or division.

4. All conditionals are evaluated, yielding a set of enabled constraints.

5. All enabled constraints are solved (deterministically, we are assuming). In the explicit case, this means that all enabled assignments are "executed" in parallel, yielding a resultant state X''.

9.5.3 Examples

To begin with, consider analog algorithms that are purely flow, that is to say without any jumps. Flow programs invoke a time parameter, which we assume is supplied by the input signal. In simple continuous-time systems, the state evolves continually, governed by ordinary differential equations, say.

For example, the motion of an idealized simple pendulum is governed by the second-order differential equation

$$\theta'' + \frac{g}{L}\theta = 0\,,$$

where θ is angular displacement, g is gravitational acceleration, and L is the length of the pendulum rod. Let the signal $u \in \mathcal{U}$ be just real time. States report the current angle $\theta \in \mathcal{F}$. All states are endowed with the same (or isomorphic) operations for real arithmetic, including sine and square root, interpreting standard symbols. Initial states contain values for g, L, and the initial angle θ_0 when the pendulum is released.

For small θ_0, the flow trajectory $\tau_t(X)$ can be specified simply by

$$\theta \;=\; \theta_0 \cdot \sin\left(\sqrt{\frac{g}{L}} \cdot \imath\right)\,,$$

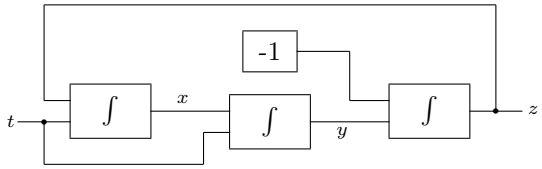


Figure 9.1: A GPAC for sine and cosine.

where $\imath$ is the input port and nothing but θ changes from state to state. The update function is, accordingly,

$$\Delta_t(X) \quad = \quad \left\{\theta \mapsto \theta_0 \cdot \sin\left(\sqrt{\frac{g}{L}} \cdot \imath\right)\right\} .$$

Hence, the critical term is $\theta_0 \cdot \sin(\sqrt{g/L} \cdot \imath)$. It can be described by the program

$$\left[\theta \textbf{ s.t. } \theta = \theta_0 \cdot \sin\left(\sqrt{\frac{g}{L}} \cdot \imath\right)\right] .$$

One of the most famous models of analog computations is the General Purpose Analog Computer (GPAC) of Claude Shannon [101]. Figure 9.1 depicts a (non-minimal) GPAC that generates sine and cosine: in this picture, the $\int$ signs denote some integrator, and the -1 denotes some constant block. If initial conditions are set up correctly, such a system will evolve according to the following initial value problem:

$$\begin{cases} x' = z & x(0) = 1 \\ y' = x & y(0) = 0 \\ z' = -y' & z(0) = 0\,. \end{cases}$$

It follows that $x(t) = \cos(t)$, $y(t) = \sin(t)$, $z = -\sin(t)$. In other words, this simple GPAC that generates sine and cosine can be modeled by program

$$[x, y, z \textbf{ s.t. } x = \cos(\imath) \wedge y = \sin(\imath) \wedge z = -y] .$$

This system could also be modeled implicitly as:

$$\mathit{Solve}\left(\{x' = z; y' = x; z' = y\}, \{x = 1; y = 0; z = 0\}, t\right) ,$$

with states incorporating an operation *Solve* that takes a system of differential equa-

tions, initial conditions, and a given time t, and returns the current values of the dynamic variables (x, y, z, in this example).

Our proposed model can also adequately describe systems (like a bouncing ball) in which the dynamics change periodically. The physics of a bouncing ball are given by the explicit flow equations

$$\begin{aligned} v &= v_0 - g \cdot t \\ x &= v \cdot t \,, \end{aligned}$$

where g is the gravitational constant, v_0 is the velocity when last hitting the table, and t is the time signal—except that upon impact, each time $x = 0$, the velocity changes according to

$$v_0 = -k \cdot v \,,$$

where k is the coefficient of impact. The critical Boolean term is $x = 0$. In any finite time interval, this condition changes value only finitely many times.

This system can be described by a program like

$$\begin{aligned} &[\ \textbf{if}\ x \neq 0\ \textbf{then}\ x, v\ \textbf{s.t.}\ v = v_0 - g \cdot t, x = (v_0 - g \cdot t) \cdot t \\ &\|\ \textbf{if}\ x = 0\ \textbf{then}\ v_0 := -k \cdot v\ ]\,, \end{aligned}$$

where x stands for its height, and v, its speed. Every time the ball bounces, its speed is reduced by a factor k.

Chapter 10

Conclusions and Future Work

In 2000, Gurevich [62] proposed an axiomatization of arbitrary (classical) algorithms. Later, Boker and Dershowitz [15, 16] provided an axiomatization of effectiveness, and proved [15, 41] that the classical version of the "Church-Turing Thesis" follows from those axioms. We have extended their axiomatization of effectiveness to the *relative* case (Section 3.5). We defined a generic measure of input size and time complexity for arbitrary domains (Definitions 26 and 27) and used that to prove the Invariance Thesis for classical algorithms (Theorem 32). We have also suggested a generic definition of a universal machine over *arbitrary* domains (Definition 11) and addressed the commonly overlooked issue of "honesty" of representation.

In [10], Blass, Gurevich, and Shelah defined classical algorithms over *unordered* domains. As an effective domain has, by its nature, an order that is not presupposed by the algorithm, unordered domains provide a more honest way to define algorithms. Inspired by that and the general definition of causal graph dynamics of Arrighi and Dowek [2], we proved that a slightly enhanced version of their model (Section 8.2.1) can simulate all classical algorithms over unordered domains (Section 8.3).

In [8], Blass and Gurevich suggested an axiomatization of generic parallel algorithm. Though their definition is very general, it is hard to restrict to the effective case. So we proposed a new model of generic parallel computation (Definition 19). Our model is less general, since it only uses shared memory interaction and bounds the number of children begotten by one agent in one step. On the other hand, this model, in its unrestricted version, allows any number of processes in the initial state (also infinitely many) and may be naturally restricted to the effective case (Section 7.5). We used this model to prove that polynomial parallel time and PSPACE are equivalent (Theorem 40).

We also gave a formalization of some desiderata of analog algorithms (Chapter 9) and generic interactive algorithms (Chapter 5).

There are various potential directions for future research. Parallel algorithms should be generalized to incorporate message passing between cells. Distributed algorithms and an appropriate ASM language were described in [61]; yet there is no proof of equivalence of the concurrent model and a generic programming language. There is also the question of what axioms may be needed for effectiveness of distributed algorithms. For the analog world, a full and satisfactory axiomatic characterization is still required. Also, a notion of (relative) effectiveness for continuous time should be devised. A description of cellular automaton for parallel, distributed, and analog cases remains to be done.

A representation of an effective domain may hide information, providing the model with unexpected computational powers. For example, in the graph domain, nodes may be ordered in a Hamiltonian path order whenever such order exists, allowing some NP problems to be solved in polynomial time. Here, we have suggested one way to overcome this difficulty by using another computational model, namely, dynamic cellular automata. Still, the proper definition of an "honest" representation of an effective domain seems elusive.

Bibliography

[1] Scott Aaronson. Quantum computing since Democritus. Lecture notes, Fall 2006. Available at http://www.scottaaronson.com/democritus/lec4.html (accessed on December 31, 2013). 7.1

[2] Pablo Arrighi and Gilles Dowek. Causal graph dynamics. *Information and Computation*, 223:78–93, February 2013. 8.1, 8.2.1, 10

[3] Mike Barnett and Wolfram Schulte. The ABCs of specification: AsmL, behavior, and components. *Informatica (Slovenia)*, 25(4):517–526, November 2001. Available at http://research.microsoft.com/pubs/73061/TheABCsOfSpecification(Informatica2001).pdf (accessed on December 31, 2013). 2.1

[4] Andreas Blass, Nachum Dershowitz, and Yuri Gurevich. When are two algorithms the same? *Bulletin of Symbolic Logic*, 15(2):145–168, 2009. Available at http://nachum.org/papers/WhenAreTwo.pdf (accessed on December 31, 2013). 2.2.4

[5] Andreas Blass, Nachum Dershowitz, and Yuri Gurevich. Exact exploration and hanging algorithms. In *Proceedings of the 19th EACSL Annual Conferences on Computer Science Logic (Brno, Czech Republic)*, volume 6247 of *Lecture Notes in Computer Science*, pages 140–154, Berlin, Germany, August 2010. Springer. Available at http://nachum.org/papers/HangingAlgorithms.pdf (accessed on December 31, 2013); longer version at http://nachum.org/papers/ExactExploration.pdf (accessed on December 31, 2013). 1.9, 2.2, 2.2.2, 2.4, 3.2, 4., 7.1, 7.3.3, 9.1, 9.3.1, 9.3.1.1, 3.

[6] Andreas Blass and Yuri Gurevich. Abstract state machines capture parallel algorithms. *ACM Transactions on Computation Logic*, 4:578–651, November 2003. Available at http://research.microsoft.com/en-us/um/people/gurevich/Opera/157-1.pdf (accessed on December 31, 2013). 1.6

[7] Andreas Blass and Yuri Gurevich. Ordinary interactive small-step algorithms, Part I. *ACM Transactions on Computational Logic*, 7(2):363–419, April 2006. Available at http://tocl.acm.org/accepted/blass04.ps (accessed on December 31, 2013). 2.2, 2.3.2

[8] Andreas Blass and Yuri Gurevich. Abstract state machines capture parallel algorithms: Correction and extension. *ACM Transactions on Computation Logic*, 9(3), June 2008. article 19. Available at http://research.microsoft.com/en-us/um/people/gurevich/Opera/157-2.pdf (accessed on December 31, 2013). 1.6, 10

[9] Andreas Blass, Yuri Gurevich, and Saharon Shelah. Choiceless polynomial time. *Annals of Pure and Applied Logic*, 100:141–187, 1999. 1.8

[10] Andreas Blass, Yuri Gurevich, and Saharon Shelah. On polynomial time computation over unordered structures. *Journal of Symbolic Logic*, 67(3):1093–1125, 2002. Available at http://research.microsoft.com/en-us/um/people/gurevich/Opera/150.pdf (accessed on December 31, 2013). 10

[11] Lenore Blum, Mike Shub, and Steve Smale. On a theory of computation and complexity over the real numbers: NP completeness, recursive functions and universal machines. *Bull. Amer. Math. Soc. (NS)*, 21:1–46, 1989. 1.3, 9.1, 9.3.1.1

[12] Udi Boker and Nachum Dershowitz. How to compare the power of computational models. In S. Barry Cooper, Benedikt Löwe, and Leen Torenvliet, editors, *Computability in Europe 2005: New Computational Paradigms (Amsterdam, The Netherlands)*, volume 3526 of *Lecture Notes in Computer Science*, pages 54–64, Berlin, Germany, 2005. Springer. 3.5

[13] Udi Boker and Nachum Dershowitz. Abstract effective models. In M. Fernández and I. Mackie, editors, *New Developments in Computational Models: Proceedings of the First International Workshop on Developments in Computational Models (DCM 2005; Lisbon, Portugal; July 2005)*, volume 135/3 of *Electronic Notes in Theoretical Computer Science*, pages 15–23, 2006. 1.9

[14] Udi Boker and Nachum Dershowitz. Comparing computational power. *Logic Journal of the IGPL*, 14(5):633–648, 2006. 1.5, 4.3

[15] Udi Boker and Nachum Dershowitz. The Church-Turing thesis over arbitrary domains. In Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich, editors, *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, volume 4800 of *Lecture Notes in Computer Science*, pages 199–229. Springer, Berlin, 2008. Available at http://nachum.org/papers/ArbitraryDomains.pdf (accessed on December 31, 2013). 1.5, 3.1, 3.2, 3, 3.2, 3.5, 4.1, 7.1, 7.1, 7.2, 10

[16] Udi Boker and Nachum Dershowitz. Three paths to effectiveness. In Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig, editors, *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, volume 6300 of *Lecture Notes in Computer Science*, pages 36–47, Berlin, Germany, August 2010. Springer. Available at http://nachum.org/papers/ThreePathsToEffectiveness.pdf (accessed on December 31, 2013). 1.9, 1.9, 3.3, 3.5, 10

[17] Egon Börger. The origins and the development of the ASM method for high level system design and analysis. *Journal of Universal Computer Science*, 8(1):2–74, 2002. Available at http://www.jucs.org/jucs_8_1/the_origins_and_the/Boerger_E.pdf (accessed on December 31, 2013). 2.1

[18] A. Borodin. On relating time and space to size and depth. *SIAM J. Comput.*, 6:733–744, 1977. Available at http://www.cs.toronto.edu/~bor/Papers/relating-time-space-size-depth.pdf (accessed on August 13, 2013). 7.1

[19] O. Bournez and M.L. Campagnolo. A survey on continuous time computations. *New Computational Paradigms. Changing Conceptions of What is Computable (Cooper, S.B., Löwe, B., Sorbi, A., eds.) New York, Springer-Verlag*, pages 383–423, 2008. 1.3, 9.1

[20] Olivier Bournez, Manuel L. Campagnolo, Daniel S. Graça, and E. Hainry. Polynomial differential equations compute all real computable functions on computable compact intervals. *Journal of Complexity*, 23:317–335, 2007. 1.3

[21] Olivier Bournez, Nachum Dershowitz, and Evgenia Falkovich. Towards an axiomatization of simple analog algorithms. In Manindra Agrawal, S. Barry Cooper, and Angsheng Li, editors, *Proceedings of the 9th Annual Conference on Theory and Applications of Models of Computation (TAMC 2012, Beijing, China)*, volume 7287 of *Lecture Notes in Computer Science*, pages 525–536, Berlin, May 2012. Springer. Available at http://nachum.org/papers/SimpleAnalog.pdf (accessed on December 31, 2013). 1.9

[22] Vannever Bush. The differential analyzer. *Journal of the Franklin Institute*, 212(4):447–488, 1931. 1.3, 9.1

[23] Samuel R. Buss, Alexander A. Kechris, Anand Pillay, and Richard A. Shore. The prospects for mathematical logic in the twenty-first century. *Bulletin of Symbolic Logic*, 7(2):1169–1196, 2001. Available at http://www.math.ucla.edu/~asl/bsl/0702/0702-001.ps (accessed on Dec. 13, 2011). 3.1

[24] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981. 7.1

[25] Alonzo Church. Review of Alan M. Turing: On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936. Journal of Symbolic Logic, vol. 2, 1937, pp. 42–43. 3.1

[26] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936. 1.5, 3.1

[27] J. Cohen and A. Slissenko. On implementations of instantaneous actions real-time asm by asm with delays. *Proceedings of the 12th Intern. Workshop on Abstract State Machines (ASM '2005) Paris, France*, pages 387–396, 2005. 9.1

[28] Joelle Cohen and Anatol Slissenko. Implementation of sturdy real-time abstract state machines by machines with delays'. *Proceedings of the 6th Intern. Conf. on Computer Science and Information Technology (CSIT 2007), Yerevan, Armenia*, September 2007. 9.1

[29] Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. *Journal of Computer Systems Science*, 7:354–375, 1973. Available at http://www.cs.utoronto.ca/~sacook/homepage/rams.pdf (accessed on December 31, 2013). 1.8, 7.1, 7.3.1, 7.3.1, 7.3.4, 7.3.4, 7.4

[30] D. Coward. Doug Coward's Analog Computer Museum, 2006. http://www.cowardstereoview.com/analog/ (accessed on Jan. 10, 2012). 9.1

[31] Martin Davis. The definition of universal Turing machine. *Proceedings Amer. Math. Soc.*, **8**:1125–1126, 1957. 1.7, 4.1

[32] Martin Davis. The myth of hypercomputation. In Christof Teuscher, editor, *Alan Turing: Life and Legacy of a Great Thinker*, pages 195–212. Springer, 2003. 3.1

[33] Nachum Dershowitz. The generic model of computation. In *Proceedings of the Seventh International Workshop on Developments in Computational Models (DCM 2011, July 2012, Zürich, Switzerland)*, Electronic Proceedings in Theoretical Computer Science, pages 59–71, 2012. Available at http://nachum.org/papers/Generic.pdf (accessed on December 31, 2013). 2.1

[34] Nachum Dershowitz. Res publica: The universal model of computation. *Proceedings of the 22nd EACSL Conference on Computer Science Logic (CSL), Torino, Italy*, pages 5–10, 2013. Available at http://www.cs.tau.ac.il/~nachumd/papers/ResPublica.pdf. 1.9

[35] Nachum Dershowitz and Evgenia Falkovich. The invariance thesis. *Logical Methods in CS*. Submitted. Available at http://www.cs.tau.ac.il/~nachumd/papers/TIH.pdf. 1.9

[36] Nachum Dershowitz and Evgenia Falkovich. A formalization and proof of the Extended Church-Turing Thesis. In *Proceedings of the Seventh International Workshop on Developments in Computational Models (DCM 2011)*, volume 88 of *Electronic Proceedings in Theoretical Computer Science*, pages 72–78, Zürich, Switzerland, July 2011. Available at http://nachum.org/papers/ECTT_EPTCS.pdf (accessed on December 31, 2013). 1.9, 7.4.0.1, 7.4.0.1, 8.3.1

[37] Nachum Dershowitz and Evgenia Falkovich. Effectiveness. In Hector Zenil, editor, *A Computable Universe*, pages 77–97. World Scientific, Singapore, December 2012. Available at http://nachum.org/papers/Universe.pdf (accessed on February 14, 2013). 1.9

[38] Nachum Dershowitz and Evgenia Falkovich. Honest universality. *Special issue of the Philosophical Transactions of the Royal Society A*, 370(1971):3340–3348, 2012. Available at http://www.cs.tau.ac.il/~nachumd/papers/HonestUniversality.pdf. 1.9

[39] Nachum Dershowitz and Evgenia Falkovich. Cellular automata are generic. *Proceedings of the Tenth International Workshop on Developments in Computational Models (DCM 2014), Ugo Dal Lago and Russ Harmer, eds., Vienna, Austria*, 2014. Available at http://www.cs.tau.ac.il/~nachumd/papers/Cell.pdf. 1.9

[40] Nachum Dershowitz and Evgenia Falkovich. Generic parallel algorithms. *Proceedings of Computability in Europe 2014: Language, Life, Limits (CiE), Arnold Beckmann, Erzsbet Csuhaj-Varj, Klaus Meer, eds., Budapest, Hungary, Lecture Notes in Computer Science*, 2014. Available at http://www.cs.tau.ac.il/~nachumd/papers/GenericParallel.pdf. 1.9

[41] Nachum Dershowitz and Yuri Gurevich. A natural axiomatization of computability and proof of Church's Thesis. *Bulletin of Symbolic Logic*, 14(3):299–350, September 2008. Available at http://nachum.org/papers/Church.pdf (accessed on December 31, 2013). 1.5, 1.9, 2.2.3, 3.1, 3.2, 7.1, 7.1, 10

[42] S. Dexter, P. Doyle, and Y. Gurevich. Gurevich abstract state machines and Schönhage storage modification machines. *J. UCS*, 1:279–303, 1997. Available at http://jucs.org/jucs_3_4/gurevich_abstract_state_machines/Dexter_S.pdf (accessed on February 14, 2013). 4., 7.1, 7.4

[43] Scott Dexter, Patrick Doyle, and Yuri Gurevich. Gurevich abstract state machines and schoenhage storage modification machines. *Springer J. of Universal Computer Science*, 3:279–303, 1997. 7.5

[44] Gilles Dowek. The physical Church thesis as an explanation of the Galileo thesis. *Natural Computing*, 11(2):247–251, 2012. 8.1

[45] D. M. Eckstein. Simultaneous memory access. *Technical Report TR-79-6, Computer Science Department, University of Iowa*, 1979. 7.3.4, 7.6

[46] Marie Ferbus-Zanda and Serge Grigorieff. ASMs and operational algorithmic completeness of lambda calculus. In Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig, editors, *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, volume 6300 of *Lecture Notes in Computer Science*, pages 301–327. Springer, Berlin, Germany, August 2010. Available at http://arxiv.org/pdf/1010.2597v1.pdf (accessed on February 14, 2013). 7.1

[47] S. Fortune and J. Wyllie. Parallelism in random access machines. *Proceedings 10th Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978. 1.6, 7.1, 7.3.4, 7.6

[48] Edward Fredkin and Tommaso Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21:219–253, 1982. 1.2

[49] Harvey M. Friedman. Mathematical logic in the 20th and 21st centuries. FOM mailing list. April 27, 2000. Available at http://cs.nyu.edu/pipermail/fom/2000-April/003913.html (accessed on December 6, 2011). 3.1

[50] A. Fröhlich and J. C. Shepherdson. Effective procedures in field theory. *Philosophical Transactions of the Royal Society of London*, 248:407–432, 1956. 3.5

[51] Zvi Galil and Wolfgang J. Paul. An efficient general-purpose parallel computer. *Journal of the ACM*, 30:360–387, 1983. 1.6

[52] Robin Gandy. Church's thesis and principles for mechanisms. In *The Kleene Symposium*, volume 101 of *Studies in Logic and the Foundations of Mathematics*, pages 123–148. North-Holland, 1980. 1.5, 2.2, 3.1, 8.1

[53] Andreas Glausch and Wolfgang Reisig. An ASM-characterization of a class of distributed algorithms. In Jean-Raymond Abrial and Uwe Glässer, editors, *Rigorous Methods for Software Construction and Analysis*, volume 5115 of *Lecture Notes in Computer Science*, pages 50–64. Springer, Berlin, 2009. Available at http://www2.informatik.hu-berlin.de/top/download/publications/GlauschR2007_dagstuhl.pdf (accessed on December 31, 2013). 9.2.2

[54] Kurt Gödel. On undecidable propositions of formal mathematical systems. *Lecture notes by S. C. Kleene and J. B. Rosser, Inst. for Advanced Study, Princeton*, 1934. Reprinted with corrections and postscriptum in M. Davis (ed.): *The Undecidable – Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*, Raven Press, 1965, pp. 39–74. The postscriptum is also reprinted in Gödel's *Collected Works*, vol. I, pp. 369–371. 3.1, 3.5

[55] E. Mark Gold. Limiting recursion. *J. Symbolic Logic*, 30(1):28–48, 1965. 2.2.1

[56] Leslie M. Goldschlager. A universal interconnection pattern for parallel computers. *Journal of the ACM*, 29:1073–1086, 1982. 1.6

[57] Saul Gorn. Algorithms: Bisection routine. *Communications of the ACM*, 3(3):174, 1960. 2.1

[58] Daniel S. Graça. Some recent developments on Shannon's general purpose analog computer. *Mathematical Logic Quarterly*, 50:473–485, 2004. 1.3

[59] Daniel S. Graça and J. Félix Costa. Analog computers and recursive functions over reals. *Journal of Complexity*, 19:644–664, 2003. 1.3

[60] Daniel Graupe. *Principles of Artificial Neural Networks*. World Scientific, 2007. 1.3

[61] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, Oxford, 1995. Available at http://research.microsoft.com/~gurevich/opera/103.pdf (accessed on December 31, 2013). 1.9, 2.1, 2.3.1, 2.3.2, 9.1, 10

[62] Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000. Available at http://research.microsoft.com/~gurevich/opera/141.pdf (accessed on December 31, 2013). 1.1, 1.9, 1.9, 2.1, 2.2, 2.2.3, 2, 6.2.2, 6.2.2, 6.2.4, 6.4, 4., 8.1, 8.3.2, 9.4.1, 10

[63] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic essence of AsmL. *Theoretical Computer Science*, 343(3):370–412, October 2005. Available at http://research.microsoft.com/~gurevich/opera/169.pdf (accessed on December 31, 2013). 2.1

[64] Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Toward industrial strength abstract state machines. Technical Report MSR-TR-2001-98, Microsoft Research, October 2001. Available at http://research.microsoft.com/en-us/um/people/gurevich/opera/155.pdf (accessed on December 31, 2013). 2.3.1

[65] Yuri Gurevich and Tatiana Yavorskaya. On bounded exploration and bounded nondeterminism. Technical report, Microsoft Research, January 2006. Available at http://research.microsoft.com/~gurevich/opera/177.pdf. 9.2.2

[66] David Harel. On folk theorems. *Communications of the ACM*, 23(7):379–389, July 1980. 2.1

[67] Juris Hartmanis. Computational complexity of random access stored program machines. *Mathematical Systems Theory*, 5:232–245, 1971. 1.8, 7.1

[68] A. Hemmerling. Systeme von Turing-Automaten und Zellularraume auf rahmbaren Pseudomustermengen. *Journal of Information Processing and Cybernetics EIK*, pages 47–72, 1979. 1.6

[69] F. E. Hennie and R. E. Stearns. Two-way simulation of multitape Turing machines. *J. of the Association of Computing Machinery*, 13:533–546, 1966. 7.1

[70] David Hilbert. Mathematische probleme: Vortrag, gehalten auf dem internationalen Mathematiker-Kongreß zu Paris 1900 (in German), 1900. Available at http://wikilivres.info/wiki/Mathematische_Probleme (accessed on Dec. 1, 2011). 1.5, 3.1

[71] David Hilbert and Wilhelm Ackermann. Grundzüge der theoretischen Logik (in German). *Springer-Verlag, Berlin*, 1920. English version of the second (1938) edition: Principles of Theoretical Logic (R. E. Luce, translator and editor), AMS Chelsea Publishing, New York, 1950. 1.5, 3.1

[72] H. J. Hoover, M. M. Klawe, and N. J. Pippenger. Bounding fan-out in logical networks. *JAMC*, 31:13–18, 1984. 7.6

[73] Stephen C. Kleene. Recursive predicates and quantifiers. *Transactions of the American Mathematical Society*, 53(1):41–73, 1943. Reprinted in M. Davis (ed.), *The Undecidable*, Raven Press, Hewlett, NY, 1965, pp. 255–287. 3.1

[74] Stephen C. Kleene. *Introduction to Metamathematics*. D. Van Nostrand, New York, 1952. 3.1, 3.5, 7.1

[75] Stephen C. Kleene. *Mathematical Logic*. Wiley, New York, 1967. 3.1

[76] Stephen C. Kleene. Reflections on Church's thesis. *Notre Dame Journal of Formal Logic*, 28(4):490–498, 1987. 2.2, 7.1

[77] Donald Knuth. Ancient Babylonian algorithms. *Communications of the ACM*, 15, 1972. Available at http://steiner.math.nthu.edu.tw/disk5/js/computer/1.pdf (accessed on September 27, 2014). 1.1

[78] Donald E. Knuth. Algorithm and program: Information and date. *Communications of the ACM*, 9:654, 1966. 1.1

[79] Andreĭ N. Kolmogorov. O ponyatii algoritma [on the concept of algorithm] (in Russian). *Uspekhi Matematicheskikh Nauk* [Russian Mathematical Surveys], 8(4):1175–1176, 1953. English version in: Vladimir A. Uspensky and Alexei L. Semenov, *Algorithms: Main Ideas and Applications*, Kluwer, Norwell, MA, 1993, pp. 18–19. 1.5, 3.1

[80] Andreĭ N. Kolmogorov and Vladimir A. Uspensky. K opredeleniu algoritma (in Russian). *Uspekhi Matematicheskikh Nauk* [Russian Mathematical Surveys], 13(4):3–28, 1958. English version: On the definition of an algorithm, *American Mathematical Society Translations*, ser. II, vol. 29, 1963, pp. 217–245. 1.5, 3.1

[81] Bruce J. MacLennan. Analog computation. *Encyclopedia of Complexity and System Science, Robert A. Meyers et al., eds., Springer*, pages 271–294, 2009. A draft is available at http://www.cs.utk.edu/~mclennan/papers/RAC-TR.pdf. 1.3

[82] A.I. Mal'tsev. Constructive algebras I. *Russian Mathematical Surveys*, 16:77–129, 1961. 3.5

[83] L. F. Menabrea. Sketch of the Analytical Engine invented by Charles Babbage. *Bibliothèque Universelle de Genève*, 82, October 1842. With notes upon the Memoir by the Translator Ada Augusta, Countess of Lovelace. 1.2

[84] Jonathan W. Mills. The nature of the extended analog computer. *Physica D: Nonlinear Phenomena*, 237:1235–1256, 2008. 1.3

[85] Ian Parberry. Parallel speedup of sequential machines: A defense of parallel computation thesis. *SIGACT News*, 18(1):54–67, March 1986. 1.8, 7.1, 7.1

[86] Detlef Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*, volume volume 2, chapter 1, pages 3–61. World Scientific, 1999. Available at http://www.informatik.uni-bremen.de/agbkb/lehre/rbs/texte/Termgraph-rewriting.pdf (accessed on December 31, 2013). 7.4.0.1, 8.3.1

[87] Emil L. Post. Absolutely unsolvable problems and relatively undecidable propositions: Account of an anticipation. In M. Davis, editor, *Solvability, Provability, Definability: The Collected Works of Emil L. Post*, pages 375–441. Birkhaüser, Boston, MA, 1994. unpublished paper, 1941. 2.2, 3.2

[88] Marian B. Pour-El and J. Ian Richards. Computability in analysis and physics. *Perspectives in Mathematical Logic*, 1, 1989. 1.3

[89] Hilary Putnam. Trial and error predicates and the solution to a problem of Mostowski. *J. Symbolic Logic*, 30(1):49–57, 1965. 2.2.1

[90] Michael. O. Rabin. Computable algebra, general theory and theory of computable fields. *Transactions of the American Mathematical Society*, 95(2):341–360, 1960. 3.5

[91] Wolfgang Reisig. On Gurevich's theorem on sequential algorithms. *Acta Informatica*, 39(4):273–305, April 2003. Available at http://www2.informatik.hu-berlin.de/top/download/publications/Reisig2003_ai395.pdf (accessed on December 31, 2013). 2.1, 6.4, 6.4, 6.4

[92] Wolfgang Reisig. The computable kernel of Abstract State Machines. *Theoretical Computer Science*, 409(1):126–136, December 2008. Draft available at http://www2.informatik.hu-berlin.de/top/download/publications/Reisig2004_hub_tr177.pdf (accessed on December 31, 2013). 3.2, 3.3, 3.4, 7.4.0.1, 9.1

[93] John Michael Robson. Random access machines with multi-dimensional memories. *Information Processing Letters*, 34:265–266, 1990. 7.1, 28, 7.3.1

[94] Hartley Rogers, Jr. On universal functions. *Proceedings of the American Mathematical Society*, **16**(1):39–44, February 1965. Available at http://www.jstor.org/stable/2033997. 1.7, 4.1

[95] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1966. 2.2.1, 3.1, 4.4

[96] Lee A. Rubel. A survey of transcendentally transcendental functions. *American Mathematical Monthly*, 96:777–788, 1989. 1.3

[97] Lee A. Rubel. The extended analog computer. *Advanced in Applied Mathematics*, 14:39–50, 1993. 1.3

[98] Heinrich Rust. Hybrid abstract state machines: Using the hyperreals for describing continuous changes in a discrete notation. In Wolf Zimmermann and Bernhard Thalheim, editors, *Proceedings of the 11th International Workshop on Abstract State Machines Advances in Theory and Practice (ASM 2004, Luther-stadt Wittenberg, Germany)*, volume 3052 of *Lecture Notes in Computer Science*, pages 281–233, Berlin, May 2004. Springer. 9.1

[99] A. Schönhage. Storage modification machines. *SIAM J. Computing*, 9:490–508, 1980. 7.3.3, 7.4

[100] A. Schönhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7:281–292, 1971. 7.6

[101] Claude E. Shannon. Mathematical theory of the differential analyzer. *Journal of Mathematics and Physics*, 20:337–354, 1941. 1.3, 9.5.3

[102] Stewart Shapiro. Acceptable notation. *Notre Dame Journal of Formal Logic*, 23(1):14–20, 1982. 4.3

[103] Joseph R. Shoenfield. *Recursion Theory*, volume 1 of *Lecture Notes In Logic*. Springer, Heidelberg, 1991. 3.1

[104] R. K. Shyamasundar and S. Ramesh. Real time programming: Languages, specification & verification. *World Scientific*, 2002. 1.3

[105] Wilfried Sieg. Mechanical procedures and mathematical experiences. *Mathematics and Mind (A. George, editor), Oxford University Press, Oxford*, pages 1 71–117, 1994. 1.5, 3.1

[106] Wilfried Sieg. Step by recursive step: Church's analysis of effective calculability. *Bulletin of Symbolic Logic*, 3(2), 1997. 1.5, 3.1

[107] Wilfried Sieg. Hilbert's programs: 1917–1922. *Bulletin of Symbolic Logic*, 5(1):1–44, 1999. Available at http://www.math.ucla.edu/~asl/bsl/0501/0501-001.ps (accessed on Dec. 13, 2011). 1.5, 3.1

[108] Wilfried Sieg and John Byrnes. K-graph machines: Generalizing turing's machines and arguments. In P. Hájek, editor, *Gödel 96: Logical Foundations of Mathematics, Computer Science, and Physics*, volume 6 of *Lecture Notes in Logic*, pages 1 98–119. Springer-Verlag, Berlin, 1996. 1.5, 3.1

[109] Wilfried Sieg and John Byrnes. An abstract model for parallel computations: Gandy's thesis. *The Monist*, 82(1):150–164, 1999. 1.5, 3.1

[110] Diomidis Spinellis. The Antikythera mechanism: A computer science perspective. *IEEE Computer*, 41(5):22–27, May 2008. 1.2

[111] Cliff Stoll. The curious history of the first pocket calculator. *Scientific American*, 290 (1):929, January 2004. 1.2

[112] J. V. Tucker and Jeffery I. Zucker. Abstract versus concrete computation on metric partial algebras. *ACM Transactions on Computational Logic*, 5(4):611–668, 2004. 3.5

[113] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937. Corrections in vol. 43 (1937), pp. 544–546. Reprinted in M. Davis (ed.), *The Undecidable*, Raven Press, Hewlett, NY, 1965. Available at http://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf (accessed on December 31, 2013). 1.5, 1.7, 1.9, 2.2, 3.1, 4.1, 7.1

[114] Alan M. Turing. Systems of logic based on ordinals. *Proceedings of the London Mathematical Society*, 45:161–228, 1939. 3.3, 3.5

[115] Alan M. Turing. Intelligent machinery. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7. Edinburgh University Press, 1969. Unpublished 1948 report for the National Physical Laboratory. 1.5

[116] Peter van Emde Boas. Machine models and simulations (Revised version). Technical Report CT-88-05, Institute for Language, Logic and Information, University of Amsterdam, August 1988. Available as http://www.illc.uva.nl/Research/Reports/CT-1988-05.text.pdf (accessed on December 31, 2013). 7.1

[117] Peter van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 1–66. North-Holland, Amsterdam, 1990. 7.1, 7.2, 7.3.1, 7.3.4

[118] U. Vishkin. Implementation of simultaneous memory address access in models that forbid it. *J. of Algorithms*, 4:45–50, 1983. 7.6

[119] John Von Neumann and Arthur W. Burks. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, IL, 1966. One chapter available at http://cba.mit.edu/events/03.11.ASE/docs/VonNeumann.pdf (accessed on May 15, 2014). 8.1

[120] J. Wiedermann. Parallel turing machines. Technical report, University Utrecht, 1984. 1.6

[121] Wikipedia. MONIAC computer. http://en.wikipedia.org/wiki/MONIAC_Computer (accessed on Mar. 1, 2012). 9.1

[122] Andrew C. Yao. Classical physics and the Church-Turing Thesis. *Journal of the ACM*, 77:100–105, January 2003. Available at http://eccc.hpi-web.de/eccc-reports/2002/TR02-062/Paper.pdf (accessed on December 31, 2013). 7.1, 7.7