Sequential Abstract State Machines Capture Sequential Algorithms

Yuri Gurevich

September 13, 1999
Revised February 20, 2000

MSR-TR-99-65

Microsoft Research
One Microsoft Way
Redmond, WA 98052-6399

Sequential Abstract State Machines Capture Sequential Algorithms

Yuri Gurevich
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA

We examine sequential algorithms and formulate a Sequential Time Postulate, an Abstract State Postulate, and a Bounded Exploration Postulate. Analysis of the postulates leads us to the notion of sequential abstract state machine and to the theorem in the title. First we treat sequential algorithms that are deterministic and noninteractive. Then we consider sequential algorithms that may be nondeterministic and that may interact with their environments.

Categories and Subject Descriptors: F.1.1 [**Theory of Computation**]: Models of Computation; I.6.5 [**Computing Methodologies**]: Model Development—*Modeling methodologies*

General Terms: Computation Models, Simulation, High-Level Design

Additional Key Words and Phrases: Turing's thesis, sequential algorithm, abstract state machine, sequential ASM thesis

1. INTRODUCTION

In 1982, I moved from mathematics to computer science. Teaching a programming language, I noticed that it was interpreted differently by different compilers. I wondered — as many did before me — what is exactly the semantics of the programming language? What is the semantics of a given program? In this connection, I studied the wisdom of the time, in particular denotational semantics and algebraic specifications. Certain aspects of those two rigorous approaches appealed to this logician and former algebraist; I felt though that neither approach was realistic.

It occurred to me that, in some sense, Turing solved the problem of the semantics of programs. While the "official" Turing thesis is that every computable function is Turing computable, Turing's informal argument in favor of his thesis justifies a stronger thesis: every algorithm can be simulated by a Turing machine. According to the stronger thesis, a program can be simulated and therefore given a precise meaning by a Turing machine (TM). In practice, it would be ridiculous to use TMs

On leave of absence from the University of Michigan.

to provide program semantics. TMs deal with single bits. Typically there is a huge gap between the abstraction level of the program and that of the TMs; even if you succeed in the tedious task of translating the program to the language of TMs, you will not see the forest for the trees.

Can one generalize Turing machines so that *any* algorithm, never mind how abstract, can be modeled by a generalized machine very closely and faithfully? The obvious answer seems to be no. The world of algorithms is too diverse even if one restricts attention to sequential algorithms as I did at the time. But suppose such generalized Turing machines exist. What would their states be? The huge experience of mathematical logic indicates that any kind of static mathematical reality can be faithfully represented as a first-order structure. So states could be first-order structures. The next question was what instructions should the generalized machines have? Can one get away with only a bounded set of instructions? Here I made the crucial observation: if we stick to one abstraction level (abstracting from low-level details and being oblivious to a possible higher-level picture) and if the states of the algorithm reflect all the pertinent information, then a particular small instruction set suffices in all cases. Hence "A New Thesis" [Gurevich 1985].

I thought of a computation as an evolution of the state. It is convenient to view relations as special functions. First-order structures with purely functional vocabulary are called algebras in the science of universal algebra. Accordingly the new machines were called dynamic structures, or dynamic algebras, or evolving algebras. The last name was the official name for a while, but then the evolving algebra community changed it to abstract state machines, or ASMs.

Even the original abstract state machines could be nondeterministic and could interact with the environment [Gurevich 1991]. Then parallel and multi-agent ASMs were introduced [Gurevich 1995]. Initially, ASMs were used to give dynamic semantics to various programming languages. Later, applications spread into many directions [Börger and Huggins 1998]. The ability to simulate arbitrary algorithms on their natural levels of abstraction, without implementing them, makes ASMs appropriate for high-level system design and analysis [Börger 1995; Börger 1999]. Many of the non-proprietary applications of ASMs can be found at [ASM Michigan Webpage].

One will also find a few papers on the ASM theory in the bibliography [Börger and Huggins 1998]. One relevant issue is the independence of data representation. Consider for example graph algorithms. Conventional computation models [Savage 1998] require that a graph be represented in one form or another, e.g. as a string or adjacency matrix, even in those cases when the algorithm is independent of the graph representation. Using ASMs (especially parallel ASMs) one can program such algorithms in a representation-independent way; see [Blass et al. 1999] in this connection.

This article is devoted to the original sequential ASM thesis that *every sequential algorithm can be step-for-step simulated by an appropriate sequential ASM.* An article on the ASM thesis for parallel computations is in preparation. The main thrust of the present article is the formalization of the notion of sequential algorithm. Turing's thesis was one underpinning of (and a shining model for) this work. The other underpinning was the notion of (first-order) structure [Tarski 1933]. The ASM model is an attempt to incorporate dynamics into the notion of structure.

By default, in this article, algorithms are supposed to be sequential, and only sequential ASMs will be introduced here. So far, experimental evidence seems to support the thesis. There is also a theoretical, speculative justification of the thesis. It was barely sketched in the literature (see [Gurevich 1991] for example), but, through the years, it was discussed at greater length in various lectures of mine. I attempted to write down some of those explanations in the dialog [Gurevich 1999]. This is a streamlined journal version of the justification. This article does not presuppose familiarity with ASMs.

An Overview of the Rest of the Article

In Section 2, we give a brief history of the problem of formalizing the notion of sequential algorithm. In Sections 3, 4 and 5 we formulate the Sequential Time Postulate, the Abstract State Postulate and the Bounded Exploration Postulate respectively. We argue that every sequential algorithm satisfies the three postulates. In Section 6, we analyze what it takes to simulate an arbitrary sequential algorithm. This leads us to the notion of sequential abstract state machine. The theorem in the title is derived from the three postulates.

Section 7 contains various remarks on ASMs. In Section 8, we generalize the Main Theorem to algorithms interacting with their environments. In Section 9, we argue that nondeterministic algorithms are special interactive algorithms and thus the generalization to interactive algorithms covers nondeterministic algorithms. Nevertheless, explicit nondeterminism may be useful. We proceed to generalize our treatment of deterministic sequential algorithms to boundedly nondeterministic sequential algorithms. This requires a slight change in the sequential-time postulate and an enhancement of the ASM programming language. In the appendix, we derive the bounded-exploration postulate from a seemingly weaker version of it (and from the sequential-time and abstract-state postulates).

Acknowledgements

I owe much to the ASM community and especially to Andreas Blass, Egon Börger and Dean Rosenzweig. Continuous discussions through the years with Andreas Blass were indispensable in clarifying things; Andreas's contribution was acknowledged already in [Gurevich 1985]. Egon Börger contributed greatly to the ASM theory and practice; much of ASM application work was done under his leadership. Dean Rosenzweig never failed to provide imaginative criticism. This paper benefited also from remarks by Colin Campbell, Jim Huggins, Jim Kajiya, Steven Lindell, Peter Päppinghaus, Grigore Rosu, and Margus Veanes. The editor, Krzysztof Apt, was very helpful indeed.

2. A BRIEF HISTORY OF THE PROBLEM

It is often thought that the problem of formalizing the notion of sequential algorithm was solved by Church [1936] and Turing [1936]. For example, according to Savage [1987], an algorithm is a computational process defined by a Turing machine. But Church and Turing did not solve the problem of formalizing the notion of sequential algorithm. Instead they gave (different but equivalent) formalizations of the notion of computable function, and there is more to an algorithm than the function it computes.

Remark 2.1. Both Church and Turing were interested in the classical decision problem (find an algorithm that decides whether a given first-order formula is valid), a central problem of logic at the time [Börger et al. 1996]. They used the formalization of the notion of computable function to prove the unsolvability of the classical decision problem. In particular, Turing put forward a thesis that every computable function is computable by a Turing machine. He proved that no Turing machine computes the validity of first-order formulas. By the thesis, validity is not computable. Notice that the Church-Turing formalization is liberal: a Church-Turing computable function may be incomputable in any practical sense. But their formalization makes undecidability results possible. □

Of course, the notions of algorithm and computable function are intimately related: by definition, a computable function is a function computable by an algorithm. Both Church and Turing spoke about arbitrary algorithms. By the stronger Turing thesis mentioned in the Introduction, every algorithm can be simulated by a Turing machine. Furthermore, the computational complexity experts agree that any algorithm can be simulated by a Turing machine with only polynomial slowdown. But a Turing machine may work much too long, with its head creeping back and forth on that infinite tape, in order to simulate one step of the given algorithm. A polynomial slowdown may be unacceptable.

While Turing analyzed a human computer, Kolmogorov and Uspensky [1958] arrived at their machine model (KU machines) by analyzing computations from the point of view of physics. Every bit of information has to be represented in physical space and can have only so many neighboring bits. One can think of a KU machine as a generalized Turing machine where the tape is a reconfigurable graph of bounded degree. Turing machines cannot simulate KU machines efficiently [Grigoriev 1976]. Kolmogorov and Uspensky did not formulate any thesis. In a paper dedicated to the memory of Kolmogorov, I attempted to do that for them: "every computation, performing only one restricted local action at a time, can be viewed as (not only being simulated by, but actually being) the computation of an appropriate KU machine" [Gurevich 1988]. Uspensky [1992, p. 396] agreed.

Influenced by Conway's "Game of Life", Gandy [1980] argued that Turing's analysis of human computations does not apply directly to mechanical devices. Most importantly, a mechanical device can be vastly parallel. Gandy put forward four principles which any such machine must satisfy. "The most important of these, called 'the principle of local causality', rejects the possibility of instantaneous action at a distance. Although the principles are justified by an appeal to the geometry of space-time, the formulation is quite abstract, and can be applied to all kinds of automata and to algebraic systems. It is proved that if a device satisfies the principles then its successive states form a computable sequence." Gandy's work provoked an interesting discussion in the logic community; I will address these issues in the forthcoming article on the ASM thesis for parallel computations.

Apparently unaware of KU machines, Schönhage [1980] introduced his storage modification machines closely related to pointer machines of Knuth [1968, pp. 462–463]. The Schönhage machine model can be seen as a generalization of the KU model where the graph is directed and only the out-degree of vertices is bounded. This generalization is natural from the point of view of pointer computations on

our computers. From the physical point of view, it is not so natural that one node may be accessed directly by an unbounded number of other nodes[1]. In any case, the abstraction level of Schönhage's model is higher than that of the KU model. It is unknown whether every Schönhage machine can be step-for-step simulated by a KU machine.

The random access machines of Cook and Reckhow [1973] are more powerful than Schönhage's machines. Additional computation models are found in [Savage 1998].

In applications, an algorithm may use powerful operations — matrix multiplication, discrete Fourier transform, etc. — as givens. On the abstraction level of the algorithm, such an operation is performed within one step and the trouble of actual execution of an operation is left to an implementation. Further, the state of a high-level algorithm does not have to be finite (which contradicts the first of Gandy's principles). There exist computation model with high-level abstractions. For example, the computation model of Blum, Shub, and Smale [1989] deals directly with genuine reals. High-level descriptions of parallel algorithms are developed in [Chandy and Misra 1988].

I sought a machine model (with a particular programming language) such that *any* sequential algorithm, however abstract, could be simulated step-for-step by a machine of that model. Let us call such a model universal with respect to sequential algorithms. Turing's model is universal with respect to computable functions, but not with respect to algorithms. In essence, the sequential ASM thesis is that the sequential ASM model is universal with respect to sequential algorithms. I don't know any other attempt to come up with a model of sequential algorithms which is universal with respect to sequential algorithms in that strong sense.

3. SEQUENTIAL TIME

This is the first of three sections on the formalization of the notion of sequential algorithm on a fixed level of abstraction. Here we formulate our first postulate.

3.1 Syntax

We assume informally that any algorithm A can be given by a finite text that explains the algorithm without presupposing any special knowledge. For example, if A is given by a program in some programming language PL, then the finite text should explain the relevant part of PL in addition to giving the PL program of A. We make no attempt to analyze such notations. There is already a bewildering variety of programming languages and various other notations to write algorithms, let alone notations that may or will be used. We will concentrate on the behavior of algorithms.

[1]One can also question how faithfully the KU model reflects physical restrictions. In a finite-dimensional Euclidean space, the volume of a sphere of radius n is bounded by a polynomial of n. Accordingly, one might expect a polynomial bound on the number of vertices in any vicinity of radius n (in the graph theoretic sense) of any state of a given KU machine, but in fact such a vicinity may contain exponentially many vertices. That fact is utilized in Grigoriev's paper mentioned above.

3.2 Behavior

Let A be a sequential algorithm.

POSTULATE 1 (SEQUENTIAL TIME). *A is associated with*

—*a set $\mathcal{S}(A)$ whose elements will be called* states *of A,*
—*a subset $\mathcal{I}(A)$ of $\mathcal{S}(A)$ whose elements will be called* initial states *of A, and*
—*a map $\tau_A : \mathcal{S}(A) \longrightarrow \mathcal{S}(A)$ that will be called the* one-step transformation *of A.*

The three associates of A allow us to define the runs of A.

DEFINITION 3.1. A *run* (or *computation*) of A is a finite or infinite sequence

$$X_0, X_1, X_2, \ldots$$

where X_0 is an initial state and every $X_{i+1} = \tau_A(X_i)$.

We abstract from the physical computation time. The computation time reflected in sequential-time postulate could be called logical. The transition from X_0 to X_1 is the first computation step, the transition from X_1 to X_2 is the second computation step, and so on. The computation steps of A form a sequence. In that sense the computation time is sequential.

DEFINITION 3.2. Algorithms A and B are *equivalent* if $\mathcal{S}(A) = \mathcal{S}(B)$, $\mathcal{I}(A) = \mathcal{I}(B)$ and $\tau_A = \tau_B$.

COROLLARY 3.3. *Equivalent algorithms have the same runs.*

In other words, equivalent algorithms have the same behavior. We study algorithms up to the equivalence relation. Since the behavior of an algorithm is determined by the three associates, it does not matter what the algorithm itself is.

3.3 Discussion

3.3.1 *States.* To us, states are full instantaneous descriptions of the algorithm. There is, however, a tendency to use the term state in a more restricted sense. A programmer may speak about the initial, waiting, active, etc. states of a software component even though these states are not full instantaneous descriptions of the component. We would prefer to speak instead about the initial, waiting, active, etc. *modes* of the component. It is said sometimes that Turing machines have only finitely many states. From our point of view, the set of states of a Turing machine is usually infinite. It is the finite control of the machine that has only finitely many configurations, but a state of the machine reflects not only the current control configuration but also the current tape configuration and the current position of the read/write head.

Call a state X of an algorithm A reachable if X occurs in some run of A. The set of reachable states of A is uniquely determined by $\mathcal{I}(A)$ and τ_A. We do not assume, however, that $\mathcal{S}(A)$ consists of reachable states only. Intuitively, $\mathcal{S}(A)$ is the set of *a priori* states of the algorithm A. Often it is simpler than the set of reachable states and thus more convenient to deal with. For example, a state of a Turing machine can be given by *any* string $uaqv$ where q is the current control state, a is the symbol in the currently observed cell and u, v are strings in the tape alphabet

(so that the tape is the string *uav* followed by an infinite tail of blanks). Not all of these states are reachable in general, and in fact the state reachability problem for a Turing machine may be undecidable. On the other hand, the restriction to reachable states would be fine for the purpose of this paper.

In applications, a system can be seen in many ways. It may involve different abstraction levels. Deciding what the states of the system are involves choosing one of those abstraction levels.

3.3.2 *Termination.* It is often required that an algorithm halts on all inputs; see [Savage 1987] for example. However many useful algorithms are not supposed to terminate. One mathematical example is the sieve of Eratosthenes that produces prime numbers. Accordingly we do not require that an algorithm halts on all inputs. But of course, a given algorithm A may have terminating runs. We stipulate that $\tau_A(X) = X$ if A terminates in a state X.

Alternatively, in addition to $\mathcal{S}(A)$, $\mathcal{I}(A)$ and τ_A, we could associate a set $\mathcal{T}(A)$ of final states with the algorithm A. It would be natural then to restrict τ_A to $\mathcal{S}(A) - \mathcal{T}(A)$. For the sake of simplicity, we don't do that. This will make the introduction of an active environment a little easier. Imagine that A arrives at a state X that is final as far as A is concerned, but then the environment changes the state, and the computation resumes. Was X truly final? We avoid that issue as well as the issue of intended final states versus error states. Besides, it is convenient that the transformation τ_A is total. This disregard of final states is not intrinsic to the ASM theory. We do not hesitate to make final states explicit when necessary or convenient [Blass and Gurevich 1997].

3.3.3 *Idle Moves.* Can a run can have idle moves so that $X_{i+1} = \tau_A(X_i) = X_i$? It does not matter for our purposes in this paper. Idle rules play little role in sequential computations. Once an idle move happens, it will repeat to the end, if any, of the run. For the sake of simplicity, we rule out idle moves here. (Idle rules play a more important role in multi-agent computations, but that is a different story.)

3.3.4 *Equivalence.* One may argue that our notion of the equivalence of algorithms is too fine, that a coarser equivalence relation could be appropriate. Consider, for example, the following two simple algorithms A and B. Each of them makes only two steps. A assigns 1 to x and then assigns 2 to y, while B first assigns 2 to y and then assigns 1 to x. According to our definition, A and B are not equivalent. Is this reasonable? The answer depends on one's goals. Our goal is to prove that for every algorithm there is an equivalent abstract state machine. The finer the equivalence, the stronger the theorem. A coarser equivalence would do as well; the theorem will remain true.

3.4 What's Left?

The fact that an algorithm is sequential means more than sequential time. There is something else.

Consider for example the graph reachability algorithm that iterates the following step. It is assumed that, initially, only the distinguished vertex `Source` satisfies the unary relation R.

```
do for all x, y with Edge(x, y) ∧ R(x) ∧ ¬R(y)
    R(y) := true
```

The algorithm is sequential-time but it is highly parallel, not sequential. A sequential algorithm should make only local changes, and the total change should be bounded.

The bounded-change requirement is not sufficient either. Consider the following graph algorithm which checks whether the given graph has isolated points.

```
if ∀x ∃y Edge(x, y) then Output := false
else Output := true
```

The algorithm changes only one Boolean but it explores the whole graph in one step and thus isn't sequential. A sequential algorithm should be not only bounded-change, but also bounded-exploration. Furthermore, the work performed by the algorithm during any one step should be bounded.

Some people find it convenient to speak in terms of actions. Notice that the one-step transformation of an algorithm may consist of several distinct actions. For example, a Turing machine can change its control state, print a symbol at the current tape cell, and move its head, all in one step. Is it true that every action can be split into atomic actions? If yes then it might be reasonable to require that the one-step transformation consists of a bounded number of atomic actions. We are going to address these issues.

4. ABSTRACT STATES

Until now, states were merely elements of the set of states. To address the issues raised in the previous subsection, we look at the notion of state more closely. We argue that states can be viewed as (first-order) structures of mathematical logic. This is a part of our second postulate, the abstract states postulate.

4.1 Structures

The notion of (first-order) structure is found in textbooks on mathematical logic. We use a slight modification of the classical notion [Gurevich 1991].

4.1.1 *Syntax.* A *vocabulary* is a finite collection of function names, each of a fixed arity. Some function names may be marked as *relational*. Every vocabulary contains the equality sign, and nullary names `true`, `false`, `undef`, and unary name `Boole`, and the names of the usual Boolean operations. With the exception of `undef`, all these *logic names* are relational.

Terms (more exactly *ground terms*; by default, terms are ground in this article) are defined by the usual induction. A nullary function name is a term. If f is a function name of positive arity j and if $t_1, \ldots, t_j$ are terms, then $f(t_1, \ldots, t_j)$ is a term. If the outermost function name is relational, then the term is *Boolean*.

4.1.2 *Semantics.* A *structure* X of vocabulary Υ is a nonempty set S (the *base set* of X) together with interpretations of the function names in Υ over S. Elements of S are also called elements of X. A j-ary function name is interpreted as a function from S^j to S, a *basic function* of X. We identify a nullary function with its value. Thus, in the context of a given structure, `true` means a particular element, namely the interpretation of the name `true`; the same applies to `false`

and `undef`. It is required that `true` be distinct from the interpretations of the names `false` and `undef`. The interpretation of a j-ary relation R is a function from S^j to $\{\texttt{true}, \texttt{false}\}$, a *basic relation* of X. The equality sign is interpreted as the identity relation on the base set. Think about a basic relation R as the set of tuples $\bar{a}$ such that $R(\bar{a}) = \texttt{true}$. If relation R is unary it can be viewed as a *universe*. `Boole` is (interpreted as) the universe $\{\texttt{true}, \texttt{false}\}$. The Boolean operations behave in the usual way on `Boole` and produce `false` if at least one of the arguments is not Boolean. `undef` allows us to represent intuitively-partial functions as total.

REMARK 4.1. One can stipulate that `undef` is the default value for the Boolean operations. Then Boolean operations become partial relations. While partial operations are natural, the tradition in mathematical logic is to deal with total relations. In a typed framework, which we do not consider in this article, one can have types of total relations as well as types of partial relations. □

A straightforward induction gives the value $Val(t, X)$ of a term t in a structure X whose vocabulary includes that of t. If $Val(t, X) = Val(t', X)$, we may say that $t = t'$ in X. If $t = \texttt{true}$ (resp. $t = \texttt{false}$) in X, we may say that t holds or is true (resp. fails or is false) in X.

4.1.3 *Isomorphism.* Let X and Y be structures of the same vocabulary Υ. Recall that an *isomorphism* from X onto Y is a one-to-one function ζ from the base set of X onto the base set of Y such that $f(\zeta x_1, \ldots, \zeta x_j) = \zeta x_0$ in Y whenever $f(x_1, \ldots, x_j) = x_0$ in X. Here f ranges over Υ, and j is the arity of f.

4.2 The Abstract State Postulate

Let A be a sequential algorithm.

POSTULATE 2 (ABSTRACT STATE).

—*States of A are first-order structures.*

—*All states of A have the same vocabulary.*

—*The one-step transformation τ_A does not change the base set of any state.*

—*$\mathcal{S}(A)$ and $\mathcal{I}(A)$ are closed under isomorphisms. Further, any isomorphism from a state X onto a state Y is also an isomorphism from $\tau_A(X)$ onto $\tau_A(Y)$.*

In the rest of this section, we discuss the four parts of the postulate.

4.3 States as Structures

The huge experience of mathematical logic and its applications indicates that any static mathematical situation can be faithfully described as a first-order structure. It is convenient to identify the states with the corresponding structures. Basic functions which may change from one state of a given algorithm to another are called *dynamic*; the other basic functions are *static*. Numerous examples of states as structures can be found in the ASM literature [Börger and Huggins 1998]. Here we give only two simple examples.

4.3.1 *Example: Turing Machines.* A state of a Turing machine can be formalized as follows. The base set of the structure includes the union of disjoint universes

`Control` ∪ `Alphabet` ∪ `Tape`

which is disjoint from {`true`, `false`, `undef`}.

—`Control` is (or represents) the set of states of the finite control. Each element of `Control` is distinguished, that is, has a (nullary function) name in the vocabulary. In addition, there is a dynamic distinguished element `CurrentControl` that "lives" in `Control`. In other words, we have the nullary function name `CurrentControl` whose value may change from one state to another.

—`Alphabet` is the tape alphabet. Each element of `Alphabet` is distinguished as well. One of these elements is called `Blank`.

—`Tape` can be taken to be the set of integers (representing tape cells). It comes with the unary operations `Successor` and `Predecessor`. (We assume for simplicity that the tape is infinite both ways.) There is also a dynamic nullary function name `Head` that takes values in `Tape`.

Finally, we have a unary function

`Content : Tape` → `Alphabet`

which assigns `Blank` to all but finitely many cells (and which assigns `undef` to every element outside of `Tape`).

The following self-explanatory rule reflects a Turing instruction:

```
if CurrentControl = q1 and Content(Head) = σ1 then
   do-in-parallel
      CurrentControl := q2
      Content(Head) := σ2
      Head := Successor(Head)
```

The whole program of a Turing machine can be written as a `do-in-parallel` block of rules like that. In the sequel, `do-in-parallel` is abbreviated to `par`.

4.3.2 *Example: The Euclidean algorithm.* The Euclidean algorithm computes the greatest common divisor d of two given natural numbers a and b. One step of it can be described as follows:

```
if       b = 0 then d := a
else if  b = 1 then d := 1
else
   par
      a := b
      b := a mod b
```

The base set of any state X of the algorithm includes the set of natural numbers which comes with $0, 1$ as distinguished elements and with the binary operation `mod`. In addition, there are three dynamic distinguished elements a, b and d. If $a = 12$, $b = 6$, $d = 1$ in X, then $a = 6$, $b = 0$, $d = 1$ in the next state X', and $a = 6$, $b = 0$, $d = 6$ in the following state X".

4.3.3 *Logician's Structures.* For the logician, the structures introduced in this section are first-order structures. There are other structures in logic, e.g. second-order and higher-order. Sometimes those other structures may be appropriate to represent states. Consider, for example, a graph algorithm that manipulates not only vertices but also sets of vertices. In this case, second-order structures are appropriate. However, second-order structures, higher-order structures, etc. can be seen as special first-order structures. In particular, the second-order graph structures needed for the graph algorithm can be viewed as two-sorted first-order structures where elements of the first sort are vertices and elements of the second sort are vertex sets. In addition to the edge relation on the first sort, there is also a cross-sort binary relation $\epsilon(x, y)$ expressing that a vertex x belongs to a vertex set y.

The term "first-order structure" may be misleading. It reflects the fact that the structures in question are used to give semantics to first-order logic. In the case of two-sorted graph-structures above, there is no first-order sentence that expresses that every vertex set is represented by an element of the second sort. But this does not matter for our purposes. We use first-order structures without limiting ourselves to first-order logic. This is a common practice in mathematics; think for example about graph theory, group theory or set theory.

4.4 Fixed Vocabulary

In logic, the vocabularies are not necessarily finite. In our case, by definition, vocabularies are finite. This reflects the informal assumption that the program of an algorithm A can be given by a finite text.

The choice of the vocabulary is dictated by the chosen abstraction level. In a proper formalization, the vocabulary reflects only truly invariant features of the algorithm rather than details of a particular state. In particular, the vocabulary does not change during the computation. One may think about a computation as an evolution of the initial state. The vocabulary does not change during that evolution.

Is it reasonable to insist that the vocabulary does not change during that evolution? One can imagine an algorithm that needs more and more functions or relations as it runs. For example, an algorithm that colors a graph may need more colors for larger graphs. It is natural to think of colors as unary relations. Thus we have a growing collection of unary relations. Notice, however, that the finite program of the coloring algorithm must provide a systematic way to deal with colors. For example, the colors may form an extensible array of unbounded length. Mathematically this gives rise to a *binary* relation $C(i, x)$ where the set of vertices of the i^{th} color is $\{x : C(i, x)\}$. In general, if an algorithm needs more and more j-ary functions of some kind, it may really deal with a $(j + 1)$-ary function. Alternatively, it may be appropriate to treat these j-ary functions as elements of a special universe.

There are also so-called self-modifying or "non-von-Neumann" algorithms which change their programs during the computation. For such an algorithm, the so-called program is just a part of the data. The real program changes that part of the data, and the real program does not change.

4.5 Inalterable Base Set

While the base set can change from one initial state to another, it does not change during the computation. All states of a given run have the same base set. Is this plausible? There are, for example, graph algorithms which require new vertices to be added to the current graph. But where do the new vertices come from? We can formalize a piece of the outside world and stipulate that the initial state contains an infinite naked set, the reserve. The new vertices come from the reserve, and thus the base set does not change during the evolution.

Who does the job of getting elements from the reserve? The environment. In an application, a program may issue some form of a NEW command; the operating system will oblige and provide more space. Formalizing this, we can use a special *external* function to fish out an element from the reserve. It is external in the sense that it is controlled by the environment.

Even though the intuitive initial state may be finite, infinitely many additional elements have muscled their way into the initial structure just because they might be needed later. Is this reasonable? I think so. Of course, we can abandon the idea of inalterable base set and import new elements from the outside world. Conceptually it would make no difference. Technically, it is more convenient to have a piece of the outside world inside the state.

It is not the first time that we reflect a piece of the outside world inside the structure. We assumed the structure contained (the denotations of) `true` and `false` which allowed us to represent relations as Boolean-valued functions. The intuitive state might have relations without containing their values; think about a graph for example. Similarly, we assumed that the structure contained `undef` which allowed us to represent intuitively-partial functions as total.

4.6 Up-to-Isomorphism Abstraction

The last part of the abstract-state postulate consists of two statements. It reflects the fact that we are working at a fixed level of abstraction. A structure should be seen as a mere representation of its isomorphism type; only the isomorphism type matters. Hence the first of the two statements: distinct isomorphic structures are just different representations of the same isomorphic type, and if one of them is a state of the given algorithm A then the other should be a state of A as well[2]. The details of how the given state represents its isomorphism type are of no importance. If they are, then the current abstraction level was chosen wrongly. The details that matter should be made explicit. The vocabulary and the basic functions should be readjusted.

To address the second statement, suppose that X and Y are distinct states of the algorithm A and ζ is an isomorphism from X onto Y. ζ maps the base set of X onto the base set of Y, and ζ preserves all functions of X. For example, if $f(a) = b$ in X, then $f(\zeta a) = \zeta b$ in Y. Since the base set is inalterable, the base sets of $\tau_A(X), \tau_A(Y)$ are those of X, Y respectively. The question is whether ζ preserves

[2]This, a set-theorist may point out, requires a proper class of states because any state has a proper class of isomorphic copies. The problem can be avoided by fixing some immense set and considering only structures whose elements are in this set. Alternatively $\mathcal{S}(A)$ and $\mathcal{I}(A)$ can be allowed to be proper classes. We will just ignore the problem.

the functions of $\tau_A(X)$. View Y as just another representation of X. An element x of X is represented by an element ζx of Y. But the representation of a state should not matter. To continue the example, suppose that τ_A sets $f(a)$ to c in X, so that $f(a) = c$ in $\tau_A(X)$. In the ζ-presentation of X (that is in Y), τ_A sets $f(\zeta a)$ to ζc, so that $f(\zeta a) = \zeta c$ in $\tau_A(Y)$.

5. BOUNDED EXPLORATION

5.1 States as Memories

It is convenient to think of a structure X as a memory of a kind. If f is a j-ary function name and $\bar{a}$ is a j-tuple of elements of X, then the pair $(f, \bar{a})$ is a *location*. $\text{Content}_X(f, \bar{a})$ is the element $f(\bar{a})$ in X.

If $(f, \bar{a})$ is a location of X and b is an element of X, then $(f, \bar{a}, b)$ is an *update* of X. The update $(f, \bar{a}, b)$ is *trivial* if $b = \text{Content}_X(f, \bar{a})$. To execute an update $(f, \bar{a}, b)$, replace the current content of location $(f, \bar{a})$ with b.

Two updates *clash* if they refer to the same location but are distinct. A set of updates is *consistent* if it has no clashing updates. To execute a consistent set of updates, execute simultaneously all updates in the set. To execute an inconsistent set of updates, do nothing. The result of executing an update set Δ over X will be denoted $X + \Delta$.

LEMMA 5.1. *If X, Y are structures of the same vocabulary and with the same base set, then there is a unique consistent set Δ of nontrivial updates of X such that $Y = X + \Delta$.*

PROOF . X and Y have the same locations. The desired Δ is

$\{(f, \bar{a}, b) : \quad b = \mathit{Content}_Y(f, \bar{a}) \neq \mathit{Content}_X(f, \bar{a})\}$. □

The set Δ will be denoted $Y - X$.

5.2 The Update Set of an Algorithm at a State

Let X be a state of an algorithm A. By the abstract-state postulate, X and $\tau_A(X)$ have the same elements and the same locations. Define

$$\Delta(A, X) \rightleftharpoons \tau_A(X) - X$$

so that $\tau_A(X) = X + \Delta(A, X)$.

LEMMA 5.2. *Suppose that ζ is an isomorphism from a state X of A onto a state Y of A, and extend ζ in the obvious way so that its domain contains also tuples of elements, locations, updates and update sets of X. Then $\Delta(A, Y) = \zeta(\Delta(A, X))$.*

PROOF . Use the last part (up-to-isomorphism abstraction) of the abstract-state postulate. □

5.3 The Accessibility Principle

By default, terms are ground (that is contain no variables) in this article, but this subsection is an exception.

According to the abstract-state postulate, an algorithm A does not distinguish between isomorphic states. A state X of A is just a particular implementation of its isomorphism type. How can A access an element a of X? One way is to produce

a ground term that evaluates to a in X. The assertion that this is the only way can be called the sequential accessibility principle.

One can think of other ways that A can access an element a of a state. For example, there could be Boolean terms φ and $\psi(x)$ such that φ is ground, and x is the only free variable in $\psi(x)$, and the equation $\psi(x) = \texttt{true}$ has a unique solution in every state X satisfying φ. If this information is available to A, it can evaluate φ at a given state X and then, if φ holds in X, point to the unique solution a of the equation $\psi(x) = \texttt{true}$ by producing the term $\psi(x)$. This involves a magical leap from the Boolean term $\psi(x)$ to the element a. To account for the magic, introduce a new nullary function name c for the unique solution of the equation $\psi(x) = \texttt{true}$ in the case that φ holds; otherwise c may be equal to `undef`. If we allow φ and $\psi(x)$ to have parameters, we will need to introduce a new function name of positive arity. This leads to a proper formalization of the given algorithm that does satisfy the sequential accessibility principle.

Definition 5.3. An element a of a structure X is *accessible* if $a = \mathit{Val}(t, X)$ for some ground term t in the vocabulary of X. A location $(f, \bar{a})$ is *accessible* if every member of the tuple $\bar{a}$ is accessible. An update $(f, \bar{a}, b)$ is accessible if both the location $(f, \bar{a})$ and the element b are accessible.

The accessibility principle and the informal assumption that any algorithm has a finite program indicate that any given algorithm A examines only a bounded number of elements in any state. Indeed, every element examined by A should be named by a ground term, but a finite program can mention only so many ground terms. This is not a formal proof, and we have no intention of analyzing the syntax of programs. So we do not elevate the accessibility principle to the status of a postulate, but it is a motivation for the bounded exploration postulate.

5.4 The Bounded Exploration Postulate

We say that two structures X and Y of the same vocabulary Υ *coincide* over a set T of Υ-terms if $\mathit{Val}(t, X) = \mathit{Val}(t, Y)$ for all $t \in T$. The *vocabulary* of an algorithm is the vocabulary of its states. Let A be a sequential algorithm.

Postulate 3 (Bounded Exploration). *There exists a finite set T of terms in the vocabulary of A such that $\Delta(A, X) = \Delta(A, Y)$ whenever states X, Y of A coincide over T.*

Intuitively, the algorithm A examines only the part of the given state which is given by means of terms in T. The set T itself is a *bounded-exploration witness* for A.

Example. The non-logic part of the vocabulary of an algorithm A consists of the nullary function name f, unary predicate name P and unary function name S.

A canonic state of A consists of the set of natural numbers and three additional distinct elements (called) `true, false, undef`. S is the successor relation on natural numbers. P is a subset of natural numbers. f evaluates to a natural number.

An arbitrary state of A is isomorphic to one of the canonic states. Every state of A is initial. The one-step transformation is given by the program

`if` $P(f)$ `then` $f := S(f)$

Clearly, A satisfies the sequential-time and abstract-state postulates. To show that it satisfies the bounded-exploration postulate, we need to exhibit a bounded-exploration witness. It may seem that the set $T_0 \rightleftharpoons \{f, P(f), S(f)\}$ is such a witness for A, but it is not. Indeed, let X be a canonic state of A where $f = 0$ and $P(0)$ holds. Set $a \rightleftharpoons \mathit{Val}(\mathtt{true}, \mathtt{X})$ and $b \rightleftharpoons \mathit{Val}(\mathtt{false}, \mathtt{X})$, so that $\mathit{Val}(P(0), X) = \mathit{Val}(\mathtt{true}, \mathtt{X}) = \mathtt{a}$. Let Y be the state obtained from X by reinterpreting `true` as b and `false` as a, so that $\mathit{Val}(\mathtt{true}, \mathtt{Y}) = \mathtt{b}$ and $\mathit{Val}(\mathtt{false}, \mathtt{Y}) = \mathtt{a}$. The value of $P(0)$ has not been changed: $\mathit{Val}(P(0), Y) = a$, so that $P(0)$ fails in Y. Then X, Y coincide over T_0 but

$$\Delta(X, A) \neq \emptyset = \Delta(Y, A).$$

The set $T = T_0 \cup \{\mathtt{true}\}$ is a bounded exploration witness for A.

6. ANALYSIS OF THE POSTULATES, ABSTRACT STATE MACHINES, AND THE MAIN THEOREM

Until now, the notion of sequential algorithm was informal. Now we are ready to formalize it.

DEFINITION 6.1. A *sequential algorithm* is an object A that satisfies the sequential-time, abstract-state and bounded-exploration postulates.

We analyze an arbitrary algorithm A. Let Υ be the vocabulary of A. In this section, all structures are Υ-structures and all states are states of A.

Let T be a bounded-exploration witness for A. Without loss of generality, we assume the following.

—T is closed under subterms: if $t_1 \in T$ and t_2 is a subterm of t_1 then $t_2 \in T$.

—T contains the logical terms `true, false, undef`.

Call terms in T *critical*. For every state X, the values of critical terms in X will be called *critical elements* of X.

LEMMA 6.2. *If $(f, (a_1, \ldots, a_j), a_0)$ is an update in $\Delta(A, X)$, then all elements $a_0, \ldots, a_j$ are critical elements of X.*

PROOF . By contradiction, assume that some a_i is not critical. Let Y be the structure isomorphic to X which is obtained from X by replacing a_i with a fresh element b. By the abstract-state postulate, Y is a state. Check that $\mathit{Val}(t, Y) = \mathit{Val}(t, X)$ for every critical term t. By the choice of T, $\Delta(A, Y)$ equals $\Delta(A, X)$ and therefore contains the update $(f, (a_1, \ldots, a_j), a_0)$. But a_i does not occur in Y. By (the inalterable-base-set part of) the abstract-state postulate, a_i does not occur in $\tau_A(X)$ either. Hence it cannot occur in $\Delta(A, Y) = \tau_A(Y) - Y$. This gives the desired contradiction. □

Since the set of critical terms does not depend on X, there is a finite bound on the size of $\Delta(A, X)$ that does not depend on X. Thus, A is bounded-change. Further, every update in $\Delta(A, X)$ is an atomic action. (Indeed, the vocabulary and the base set of a state do not change during the computation. Only basic functions do. To change a state in a minimal possible way so that the result is a

legal structure, change one basic function at one place, i.e., change the content of one location. That is exactly what one update does.) Thus $\Delta(A, X)$ consists of a bounded number of atomic actions.

To program individual updates of $\Delta(A, X)$, we introduce update rules.

DEFINITION 6.3. An *update rule* of vocabulary Υ has the form

$$f(t_1, \ldots, t_j) := t_0$$

where f is a j-ary function symbol in Υ and $t_0, \ldots, t_j$ are terms over Υ. To fire the update rule at an Υ-structure X, compute elements $a_i = \mathit{Val}(t_i, X)$ and then execute the update $(f, (a_1, \ldots, a_j), a_0)$ over X.

By virtue of Lemma 6.2, every update in $\Delta(A, X)$ can be programmed as an update rule. To program the whole $\Delta(A, X)$, we need a rule that allows us to execute all updates in $\Delta(A, X)$ in parallel, as a single transaction. This leads us to a `par` construct.

DEFINITION 6.4. If k is any natural number and $R_1, \ldots, R_k$ are rules of vocabulary Υ, then

```
par
    R_1
    R_2
    ⋮
    R_k
endpar
```

is a rule of vocabulary Υ. To fire the `par` rule at an Υ-structure X, fire the constituent rules $R_1, \ldots, R_k$ simultaneously.

The `par` rules are called *blocks*. The empty block (with zero constituent rules) is abbreviated to `skip`. For the purpose of programming update sets $\Delta(A, X)$, we will need only blocks with update-rule constituents, but the extra generality of Definition 6.4 will be useful to us. To give more rigorous semantics to rules, we define the update set $\Delta(R, X)$ that a rule R of vocabulary Υ generates at any Υ-structure X.

DEFINITION 6.5. If R is an update rule $f(t_1, \ldots, t_j) := t_0$ and $a_i = \mathit{Val}(t_i, X)$ for $i = 0, \ldots, j$ then

$$\Delta(R, X) \rightleftharpoons \{(f, (a_1, \ldots, a_j), a_0)\}.$$

If R is a `par` rule with constituents $R_1, \ldots, R_k$, then

$$\Delta(R, X) \rightleftharpoons \Delta(R_1, X) \cup \cdots \cup \Delta(R_k, X).$$

COROLLARY 6.6. *For every state X, there exists a rule R^X such that*

1. *R^X uses only critical terms, and*
2. *$\Delta(R^X, X) = \Delta(A, X)$.*

In the rest of this section, R^X is as in the corollary.

LEMMA 6.7. *If states X and Y coincide over the set T of critical terms, then $\Delta(R^X, Y) = \Delta(A, Y)$.*

PROOF . We have

$$\Delta(R^X, Y) = \Delta(R^X, X) = \Delta(A, X) = \Delta(A, Y).$$

The first equality holds because R^X involves only critical terms and because critical terms have the same values in X and Y. The second equality holds by the definition of R^X. The third equality holds because of the choice of T and because X and Y coincide over T. □

LEMMA 6.8. *Suppose that X, Y are states and $\Delta(R^X, Z) = \Delta(A, Z)$ for some state Z isomorphic to Y. Then $\Delta(R^X, Y) = \Delta(A, Y)$.*

PROOF . Let ζ be an isomorphism from Y onto an appropriate Z. Extend ζ to tuples, locations, updates and set of updates. It is easy to check that $\zeta(\Delta(R^X, Y)) = \Delta(R^X, Z)$. By the choice of Z, $\Delta(R^X, Z) = \Delta(A, Z)$. By Lemma 5.2, $\Delta(A, Z) = \zeta(\Delta(A, Y))$. Thus $\zeta(\Delta(R^X, Y)) = \zeta(\Delta(A, Y))$. It remains to apply ζ^{-1} to both sides of the last equality. □

At each state X, the equality relation between critical elements induces an equivalence relation

$$E_X(t_1, t_2) \rightleftharpoons \mathit{Val}(t_1, X) = \mathit{Val}(t_2, X)$$

over critical terms. Call states X, Y *T-similar* if $E_X = E_Y$.

LEMMA 6.9. *$\Delta(R^X, Y) = \Delta(A, Y)$ for every state Y T-similar to X.*

PROOF . By Lemma 6.8, it suffices to find a state Z isomorphic to Y with $\Delta(R^X, Z) = \Delta(A, Z)$.

First we consider a special case where Y is disjoint from X, that is where X and Y have no common elements. Let Z be the structure isomorphic to Y that is obtained from Y by replacing $\mathit{Val}(t, Y)$ with $\mathit{Val}(t, X)$ for all critical terms t. (The definition of Z is coherent: if t_1, t_2 are critical terms, then

$$\mathit{Val}(t_1, X) = \mathit{Val}(t_2, X) \iff \mathit{Val}(t_1, Y) = \mathit{Val}(t_2, Y)$$

because X and Y are T-similar.) By the abstract-state postulate, Z is a state. Since X and Z coincide over T, Lemma 6.7 gives $\Delta(R^X, Z) = \Delta(A, Z)$.

Second we consider the general case. Replace every element of Y that belongs to X with a fresh element. This gives a structure Z that is isomorphic to Y and disjoint from X. By the abstract-state postulate, Z is a state. Since Z is isomorphic to Y, it is T-similar to Y and therefore T-similar to X. By the first part of this proof, $\Delta(R^X, Z) = \Delta(A, Z)$. □

For every state X, there exists a Boolean term φ^X that evaluates to `true` in a structure Y if and only if Y is T-similar to X. The desired term asserts that the equality relation on the critical terms is exactly the equivalence relation E_X. Since there are only finitely many critical terms, there are only finitely many possible equivalence relations E_X. Hence there is a finite set $\{X_1, \ldots X_m\}$ of states such that every state is T-similar to one of the states X_i.

To program A on all states, we need a single rule that is applied to every state X and has the effect of R^X at X. This leads naturally to the `if-then-else` construct and to conditional rules.

DEFINITION 6.10. If φ is a Boolean term over vocabulary Υ and R_1, R_2 are rules of vocabulary Υ then

```
if φ then R1
else R2
endif
```

is a rule of vocabulary Υ. To fire R at any Υ-structure X, evaluate φ at X. If the result is `true` then $\Delta(R, X) = \Delta(R_1, X)$; otherwise $\Delta(R, X) = \Delta(R_2, X)$. □

The `else` clause may be omitted if R_2 is `skip`. Such `if-then` rules would suffice for our purposes here, but the extra generality will be useful. In this article, we will usually omit the keywords `endpar` and `endif`.

A sequential ASM *program* Π of vocabulary Υ is just a rule of vocabulary Υ. Accordingly, $\Delta(\Pi, X)$ is well defined for every Υ structure X.

LEMMA 6.11 (MAIN LEMMA). *For every sequential algorithm A of vocabulary Υ there is an ASM program Π of vocabulary Υ such that $\Delta(\Pi, X) = \Delta(A, X)$ for all states X of A.*

PROOF . Let $X_1, \ldots, X_m$ be as in the discussion following Lemma 6.9. The desired Π is

```
par
    if   φ^X1  then   R^X1
    if   φ^X2  then   R^X2
    ...
    if   φ^Xm  then   R^Xm
```

The lemma is proved. □

Nesting `if-then-else` rules gives an alternative proof of the Main Lemma. The desired Π could be

```
if         φ^X1    then   R^X1
else if    φ^X2    then   R^X2
...
else if    φ^Xm    then   R^Xm
```

Given a program Π, define

$$\tau_\Pi(X) \rightleftharpoons X + \Delta(\Pi, X).$$

DEFINITION 6.12. A *sequential abstract state machine* B of vocabulary Υ is given by

—a program Π of vocabulary Υ,
—a set $\mathcal{S}(B)$ of Υ-structures closed under isomorphisms and under the map τ_Π,
—a subset $\mathcal{I}(B) \subseteq \mathcal{S}(B)$ that is closed under isomorphisms,

—the map τ_B which is the restriction of τ_Π to $\mathcal{S}(B)$.

It is easy to see that an abstract state machine satisfies the sequential-time, abstract-state and bounded-exploration postulates. By Definition 6.1, it is an algorithm. Recall the definition of equivalent algorithms, Definition 3.2.

THEOREM 6.13 (MAIN THEOREM). *For every sequential algorithm A, there exists an equivalent sequential abstract state machine B.*

PROOF . By the Main Lemma, there exists an ASM program Π such that $\Delta(\Pi, X) = \Delta(A, X)$ for all states X of A. Set $\mathcal{S}(B) = \mathcal{S}(A)$ and $\mathcal{I}(B) = \mathcal{I}(A)$. □

7. REMARKS ON ABSTRACT STATE MACHINES

7.1 Constructivity

Traditionally it is required in the foundations of mathematics that inputs to an algorithm be constructive objects (typically strings) and that the states of an algorithm have constructive representations. One champion of that tradition was Markov [1954]. As we mentioned in Section 2, these constructivity requirements may be too restrictive in applications, especially in high-level design and specification. We abstract from the constructivity constraints. ASMs are algorithms which treat their states as databases or oracles. In that — very practical — sense, ASMs are executable. A number of non-proprietary tools for executing ASMs can be found at [ASM Michigan Webpage]. The abstraction from the constructivity requirements contributed both to the simplicity of the definition of ASMs and to their applicability.

7.2 Additional Examples of ASM Programs

Two ASM programs were given earlier in Section 4. Here are three additional examples.

7.2.1 *Maximal Interval Sum.* The following problem and the mathematical solution are borrowed from [Gries 1990]. Suppose that A is a function from $\{0, 1, \ldots, n-1\}$ to real numbers and i, j, k range over $\{0, 1, \ldots, n\}$. For all $i \leq j$, let $S(i, j) \rightleftharpoons \sum_{i \leq k < j} A(k)$. In particular, every $S(i, i) = 0$. The problem is to compute $S \rightleftharpoons \max_{i \leq j} S(i, j)$ efficiently. Define $y(k) \rightleftharpoons \max_{i \leq j \leq k} S(i, j)$. Then $y(0) = 0$, $y(n) = S$ and

$$\begin{aligned} y(k+1) &= \max\{\max_{i \leq j \leq k} S(i,j), \max_{i \leq k+1} S(i, k+1)\} \\ &= \max\{y(k), x(k+1)\} \end{aligned}$$

where $x(k) \rightleftharpoons \max_{i \leq k} S(i, k)$, so that $x(0) = 0$ and

$$\begin{aligned} x(k+1) &= \max\{\max_{i \leq k} S(i, k+1), S(k+1, k+1)\} \\ &= \max\left\{\max_{i \leq k}\Big(S(i,k) + A(k)\Big), 0\right\} \\ &= \max\left\{\Big(\max_{i \leq k} S(i,k)\Big) + A(k), 0\right\} \end{aligned}$$

$$= \max\{x(k) + A(k), 0\}$$

Since $y(k) \geq 0$, we have

$$y(k+1) = \max\{y(k), x(k+1)\} = \max\{y(k), x(k) + A(k)\}$$

Assume that nullary dynamic functions k, x, y equal zero in the initial state. The desired algorithm is

```
if k ≠ n then
   par
      x := max{x + A(k), 0}
      y := max{y, x + A(k)}
      k := k + 1
else S := y
```

7.2.2 *A Higher-Level View of Turing Machines.* Let `Alphabet, Control, Tape, Head` and `Content` be as in the example on Turing machines in Section 4. Retire `Successor` and `Predecessor`. Instead introduce a subuniverse `Displacement` = $\{-1, 0, 1\}$ of `Tape`, each element of which is distinguished, and a binary function + of type `Tape` × `Displacement` → `Tape` with the obvious meaning. Every Turing machine gives rise to finite functions

```
NewControl :  Control × Alphabet → Control
NewSymbol :   Control × Alphabet → Alphabet
Displace :    Control × Alphabet → Displacement
```

Taking these three functions (extended properly by means of the default value `undef`) as givens, leads us to the following higher-level view of Turing machines:

```
par
   CurrentControl := NewControl(CurrentControl,Content(Head))
   Content(Head) := NewSymbol(CurrentControl,Content(Head))
   Head := Head + Displace(CurrentControl,Content(Head))
```

7.2.3 *Markov's Normal Algorithms.* A normal algorithm A operates on strings of a given alphabet Σ. It is given by a sequence of productions

$$x_1 \to y_1, \ldots, x_n \to y_n$$

where every x_i and every y_i are Σ-strings. Some of the productions can be marked as final. States of A are Σ-strings. To perform one step of A at a state Z do the following. Check if any x_i is a contiguous segment of Z. If not, halt. If yes, find the least such i and find the first occurrence of x_i in Z. Let $\texttt{Pre}(x_i, Z)$ be the part of Z before the first occurrence of x_i and let $\texttt{Post}(x_i, Z)$ be the part of Z after the first occurrence of x_i, so that

$$Z = \texttt{Pre}(x_i, Z) * x_i * \texttt{Post}(x_i, Z)$$

where $*$ is the concatenation operation on strings. Replace Z with a new state

$$\texttt{Pre}(x_i, Z) * y_i * \texttt{Post}(x_i, Z).$$

If the i^{th} production is final, halt.

Normal algorithms do not satisfy Gandy's principle of local causality. "The process of deciding whether a particular substitution is applicable to a given word is essentially global" [Gandy 1980]. Let $\texttt{Occurs}(x, Z)$ be the Boolean test of whether x occurs in Z as a contiguous segment. Markov treats this test and the functions $\texttt{Pre}, \texttt{Post}$ and $*$ as givens, and so will we. Introduce a universe `String` of all strings over the alphabet Σ and stipulate that $\texttt{Pre}(x, Z) = \texttt{Post}(x, Z) = \texttt{undef}$ if $\texttt{Occurs}(x, Z) = \texttt{false}$.

It is easy to see how to program any normal algorithm over Σ. For example, the normal algorithm

$$x_1 \to y_1, x_2 \to y_2, x_3 \to y_3$$

where none of the productions is final, can be programmed thus:

```
if      Occurs(x1, Z) then Z := Pre(x1, Z) * y1 * Post(x1, Z)
else if Occurs(x2, Z) then Z := Pre(x2, Z) * y2 * Post(x2, Z)
else if Occurs(x3, Z) then Z := Pre(x3, Z) * y3 * Post(x3, Z)
```

7.3 The `let` Construct

An ASM rule R of vocabulary Υ can be seen as an algorithm all by itself. The states of R are arbitrary Υ-structures, and every state is initial. $\tau_R(X) = X + \Delta(R, X)$. Accordingly, two rules of vocabulary Υ are equivalent if and only if they generate the same update set $\Delta(X)$ at every Υ-structure X. For example,

```
if φ                              par
then R1      is equivalent to        if φ then R1
else R2                              if ¬φ then R2
```

Notice that the latter rule is more verbose, especially in the case of large φ. This brings us to the issue of more concise programs. A `let` rule

```
let x = t
    R(x)
```

saves one the trouble of writing and evaluating the term t over and over again. Here x is a fresh nullary function name that occurs neither in R nor in t. This is one example of syntactic sugar used in ASM programming. In the simplified ASM framework described above,

```
let x = t
    R(x)      is equivalent to      R(t)
```

where $R(t)$ is the result of simultaneous substitution of t for every occurrence of x in $R(x)$.

7.4 Sequential Composition and the Linear Speedup Theorem

In conventional programming languages, sequential composition allows one to combine pieces of different granularities: trees, and forests, and bushes, and grass. As a result, programs may become hard to understand. A sequential ASM program Π

describes only one computation step. This unusual programming style helps one to stick to one abstraction level[3].

In principle, however, sequential composition can be incorporated into the ASM paradigm. One step of the rule

```
seq
    R1
    R2
```

consists of two successive substeps. The second substep may overwrite some changes made during the first substep. If a clash occurs at either substep, the whole step is aborted.

It is easy to see that programs in the enriched language satisfy our three postulates, and thus `seq` can be eliminated. In fact, there is a uniform way of eliminating `seq`. Similarly one can prove the linear speedup theorem for the original ASM language:

THEOREM 7.1. *For every program* Π_1*, there is a program* Π_2 *of the same vocabulary such that* $\tau_{\Pi_2}(X) = \tau_{\Pi_1}(\tau_{\Pi_1}(X))$ *for every* Υ*-structure* X*.*

The existing ASM-executing tools avoid the `seq` construct, partially for clarity and partially for technical reasons. Most of the tools implement the powerful parallel `do-for-all` construct [Gurevich 1995; Gurevich 1997], which makes implementation of `seq` expensive. Other constructs with a flavor of sequentiality, e.g. `let`, have been more popular.

7.5 ASMs and Recursion

A common question is whether ASMs can handle recursion. We view recursive computations being implicitly multi-agent and treat them accordingly [Gurevich and Spielmann 1997].

7.6 Inconsistent Update Sets

Let Δ be an inconsistent update set over a structure X. We stipulated that $X+\Delta = X$, so that the effect of Δ is that of the empty update set. There are reasonable alternatives to the stipulation; one may want to manifest inconsistency in one way or another [Gurevich 1995]. In practice, the simple stipulation seems to work well. If desired, manifestations of inconsistency can be programmed. In any case, inconsistent update sets do not arise in the proof of the main theorem, and thus the issue is irrelevant to this paper.

7.7 Typed Abstract State Machines

First-order structures are versatile. Our version of the notion of structure supports informal typing. However, there are pragmatic reasons to introduce explicit typing; most programming languages followed this route. The issue of typed ASMs has been and is being explored. See [ASM Michigan Webpage] in this connection. You will find there ASM-executing tools (e.g. AsmGofer, ASM Workbench) that use typed versions of the ASM programming language.

[3]This unusual style may be convenient for programming highly distributed systems where an agent cannot compute much without a danger of being interrupted.

8. ALGORITHMS INTERACTING WITH THEIR ENVIRONMENTS

Until now we have considered non-interactive algorithms. Here we incorporate active environments into the picture. The definition of an algorithm as something that satisfies the three postulates (Definition 6.1) does not change. The definition of the equivalence of algorithms (Definition 3.2) does not change either. What does change is the definition of runs. Corollary 3.3 (equivalent algorithms have the same runs) will remain true. The Main Theorem will remain true.

8.1 Active Environment

8.1.1 *Example.* The following interactive version of the Euclidean algorithm repeatedly computes the greatest common divisor.

```
par
   if Mode = Initial then
      a := Input1, b := Input2, Mode := Compute
   if Mode = Compute then
      if b = 0 then d := a, Mode := Wait
      else if b = 1 then d := 1, Mode := Wait
      else a := b, b := a mod b
```

In a state with `Mode = Wait`, the algorithm waits for the environment to intervene, to change `Mode` to `Initial` and to update `Input1` and `Input2`. It is assumed that the environment satisfies the following constraints: it intervenes only when `Mode = Wait`; the only functions affected by it are `Mode, Input1` and `Input2`; it can change `Mode` only to `Initial`; `Input1` and `Input2` have to be natural numbers. The constraints allow the environment to be non-deterministic. It may be for example that the third pair of inputs equals the first pair of inputs but the fourth pair of inputs differs from the second pair.

8.1.2 *Agents.* It may be convenient to think of computations of an interactive program as plays in a two-player game where one player is the agent executing our program, let us call it α, and the other player is the active environment. In reality, many agents could be involved. Relationships among the agents may involve both cooperation and competition. Furthermore, new agents may be created and old agents may be removed. Elsewhere we deal with multiple agents explicitly [Gurevich 1995]. In the sequential paradigm, agents are implicit. The environment accounts for the actions of all outside agents (that is all agents distinct from our agent α) as well as for some global constraints.

8.2 Interactive Runs

An interactive algorithm can input and output data as it runs. We can abstract from the output phenomenon. As far as our algorithm is concerned, the output is miraculously taken away. To account for the input phenomenon, we modify the definition of run given in Section 3. A run of a non-interactive algorithm can be viewed as a sequence

$$\mu_1, \mu_2, \mu_3, \ldots$$

of non-idle moves (or non-idle steps) of the algorithm. The moves of an interactive algorithm A may be interspersed with moves of the environment. For example, A

may have a run

$$\mu_1, \nu_1, \mu_2, \nu_2, \mu_3, \nu_3, \ldots$$

where A makes the μ moves and the environment makes the ν moves, so that we have a sequence

$$X_0, X_1, X_1', X_2, X_2', X_3, X_3', \ldots$$

of states where X_0 is an initial state, $X_1 = \tau_A(X_0)$, X_1' is obtained from X_1 by an action of the environment, $X_2 = \tau_A(X_1')$, X_2' is obtained from X_2 by an action of the environment, and so on.

It may seem strange that the environment should act only between the steps of the algorithm. In Example 8.1.1, the environment may prepare new `Input1` and `Input2` during the time that the algorithm works on the old `Input1` and `Input2`. Our definition of run reflects the point of view of the executor of the given algorithm. He will not see the new inputs until he finishes with the old ones. As far as he is concerned, the new inputs appear after he finishes with the old ones. Different agents may see the same computation very differently.

One can argue that a run should not start or finish with a move of the environment, that every move by the environment should be followed by a move of the algorithm. This is reasonable but these details are immaterial for our purposes in this paper.

It is easy to see that the Main Theorem, Theorem 6.13, remains valid. There is no need to reconsider any of the postulates.

8.3 New Elements

According to the discussion on inalterable base sets in Section 4, new elements are imported by the environment and a special external function is used for the purpose. It may be convenient to hide the external function. To this end, `create` (formerly `import`) rules are used [Gurevich 1991]:

```
create x
   R(x)
```

Typically a `create` rule is used to extend some particular universe U which gives rise to the `extend U` abbreviation. For example, consider a Turing machine with finite tape where `Last` is the rightmost cell. An instruction may involve an extension of the tape:

```
extend Tape with x
   par
      Successor(Last) := x
      Predecessor(x) := Last
      Last := x
      ...
```

Instead of importing new elements from the reserve (or from the outside world for that matter), it may be convenient to deal with a virtual world that has everything that the computation may possibly need. You never create new elements but rather discover or activate them [Blass et al. 1999].

8.4 Discussion

In applications, the environment usually satisfies stringent constraints. The definition of vocabulary in [Gurevich 1995] allows one to mark function names as *static*; these functions do not change during the computation. In the example above, it would be natural to declare the nullary functions names $0, 1$, `Initial, Compute, Wait` and the binary function name `mod` static. Another typical constraint is that the environment satisfies the type discipline of a sort.

Basic dynamic functions (those that can change during the computation are partitioned into

—*internal*, those whose values can be changed only by the algorithm,

—*external*, those whose values can be changed only by the environment, and

—*shared*, those whose values can be changed both by the environment and the algorithm.

(In some applications, it is convenient to call internal functions *controlled* and external functions *monitored* [Börger 1999]). In the example above, a, b, d are internal, `Input1` and `Input2` are external, and `Mode` is shared. The same terminology applies to locations.

The executor of our algorithm may need to consult the environment in the middle of executing a step. For example, after evaluating various guards, the executor may discover that he needs the environment to create new elements and/or make some choices. Of course we may pretend that the environment did all the necessary work before the algorithm started the execution of the step. But a refinement of our definition of interactive runs may be useful. This issue will be addressed in [Gurevich 2000].

9. NONDETERMINISTIC SEQUENTIAL ALGORITHMS

9.1 How Can Algorithms be Nondeterministic?

Nondeterministic algorithms are useful, for example, as higher level descriptions (specifications) of complicated deterministic algorithms. But an algorithm is supposed to be an exact recipe for execution. How can it be non-deterministic? Let us analyze that apparent contradiction.

Imagine that you execute a nondeterministic algorithm A. In a given state, you may have several alternatives for your action and you have to choose one of the available alternatives. The program of A tells you to make a choice but gives no instructions how to make the choice. What can you do? You can improvise and, for example, flip a coin to direct the choice. You can write an auxiliary program to make choices for you. You can ask somebody to make choices for you. Whatever you do, you bring something external to the algorithm. In other words, it is the active environment that makes the choices. Monitored functions can be used to manifest the choices made by the active environment. In that sense, nondeterminism has been handled already. Nondeterministic algorithms are special interactive algorithms.

9.2 Bounded-Choice Nondeterminism

It may be convenient to pretend that the choices are made by an algorithm itself rather than by the environment. In the case of sequential algorithms where the one-step transformation is supposed to be bounded-work, it is natural to require that nondeterminism is bounded, so that, at each step, the choice is limited to a bounded number of alternatives. Nondeterministic Turing machines are such bounded-choice nondeterministic algorithms.

Modify slightly the sequential-time and abstract-state postulates to obtain the nondeterministic versions of these two postulates. Now τ_A is a binary relation over $\mathcal{S}(A)$, and the last two parts of the nondeterministic abstract-state postulate are as follows.

—If $(X, X') \in \tau_A$ then the base set of X' is that of X.

—$\mathcal{S}(A)$ and $\mathcal{I}(A)$ are closed under isomorphisms. Further, let ζ be an isomorphism from a state X onto a state Y. For every state X' with $(X, X) \in \tau_A$, there is a state Y' with $(Y, Y') \in \tau_A$ such that ζ is an isomorphism from X' onto Y'.

A *bounded-choice nondeterministic algorithm* is an object A that satisfies the nondeterministic sequential-time and abstract-state postulates and the unchanged bounded-exploration postulate. Check that, for every such algorithm A, there is a natural number k such that, for every $X \in \mathcal{S}(A)$, the set $\{Y : (X, Y) \in \tau_A\}$ has at most k members.

Extend the programming language of deterministic sequential ASMs with the `choose-among` construct:

```
choose among
    R_1
    ...
    R_k
```

Call bounded-choice nondeterministic algorithms A and B *equivalent* if $\mathcal{S}(A) = \mathcal{S}(B)$, $\mathcal{I}(A) = \mathcal{I}(B)$ and $\tau_A = \tau_B$. It is easy to see that Main Theorem remains valid in the case of bounded-choice nondeterministic algorithms.

A more powerful choose construct that may require unbounded exploration is given in [Gurevich 1995].

APPENDIX

A. FINITE EXPLORATION

Assuming the sequential-time and abstract-state postulates, we derive the bounded-exploration postulate from seemingly weaker hypotheses. To this end, we do not require in this appendix that an algorithm satisfies the bounded-exploration postulate: an algorithm is defined as any object that satisfies the sequential-time and abstract-state postulates. Logically this appendix belongs at the end of Section 5 but we deferred it to avoid interrupting the main discussion.

Definition A.1. An algorithm A of vocabulary Υ is *finite-exploration* if there is a map $\mathcal{T}$ that assigns a finite set $\mathcal{T}(X)$ of Υ-terms to each state X of A in such a way that the following two requirements are satisfied.

(1) $\Delta(A, Y) = \Delta(A, X)$ for every state Y of A that coincides with X over $\mathcal{T}(X)$.
(2) If X, Y are two states of A then either $\mathcal{T}(X) = \mathcal{T}(Y)$ or else there are terms t_1, t_2 in $\mathcal{T}(X) \cap \mathcal{T}(Y)$ whose values are equal in one of the states and different in the other.

Intuitively the first requirement is that, at each step, A examines only the finite part of the current state X which is given by the terms in $\mathcal{T}(X)$. Notice that the set $\mathcal{T}(X)$ may vary with the state X.

The second requirement reflects the determinism of A. Andreas Blass suggested the following scenario. The state is a database. At each step, the executing agent of A queries the database and computes the appropriate updates. The updates may be conveyed to the database e.g. in the form of update rules. But what can the agent learn? Since (by the abstract-state postulate) the internal presentations of the elements of the state are of no importance, the agent can learn only that such and such terms have the same value, and such and such terms have different values. The (one-step) computations of the agent at states X and Y can branch apart if the agent discovers that some terms t_1, t_2 have equal values in one of the two states and different values in the other. Otherwise the computations are identical, and therefore $\mathcal{T}(X) = \mathcal{T}(Y)$.

Call an algorithm A *bounded-exploration* if it satisfies the bounded-exploration postulate. In other words, A is bounded-exploration if and only if it is finite-exploration with a finite exploration witness $\mathcal{T}$ such that $\mathcal{T}(X)$ does not depend on X.

What happens if a finite-exploration algorithm A with a finite-exploration witness $\mathcal{T}$ is applied to an Υ-structure Y that is not a state of A? Since A and Y have the same vocabulary, the queries of the execution agent should make sense in Y. The agent may learn that such and such terms have the same value in Y, and such and such terms have different values in Y. Two possibilities arise.

—There is a state X such that Y coincides with X over $\mathcal{T}(X)$. In this case, Y could be called a *pseudo-state* of A (with respect to $\mathcal{T}$).
—There is no state X such that Y coincides with X over $\mathcal{T}(X)$. In this case, Y could be called an *error state*[4] of A (with respect to $\mathcal{T}$).

For emphasis, states of A may be called *legal states*. We are interested in algorithms which can distinguish between legal states and error states in the following sense.

Definition A.2. An algorithm A of vocabulary Υ is *discriminating* if there is a map $\mathcal{T}$ that assigns a finite set $\mathcal{T}(X)$ of Υ-terms to each Υ-structure X in such a way that the following requirements are satisfied.

—$\mathcal{T}$ is a finite-exploration witness[5] for A.
—If Y is an error state of A with respect to $\mathcal{T}$, then no legal state of A coincides with Y over $\mathcal{T}(Y)$.

[4]An error state is not a state. Such a terminology is not that uncommon. A red herring is often not a herring and not necessarily red either.
[5]Or the restriction of $\mathcal{T}$ to legal states is a finite-exploration witness. We don't have to require in Definition A.1 that the domain of $\mathcal{T}$ is limited to the states of A.

If A is a bounded-exploration algorithm with bounded-exploration witness T, then A is discriminating with $\mathcal{T}(X) \rightleftharpoons T$.

EXAMPLE A.3. Suppose that A is an algorithm of vocabulary Υ whose non-logic part consists of nullary function name 0, unary function name f and nullary relation name `parity`. An Υ-structure X is a legal state of A if the set $S(X)$ of the values of terms $0, f(0), f(f(0)), \ldots$ is finite in X. A state X of A is initial if `parity = false` in X. A performs only one step. If the cardinality of $S(X)$ is even, `parity` is set to `true`; otherwise there is no change.

It is easy to see that A is finite-exploration. However, A is not discriminating. Consider a Υ-structure Y where the terms $0, f(0), f(f(0)), \ldots$ have distinct values. Y is an error state but, for every set T of Υ-terms, there is a large enough legal state of A that coincides with Y over T. □

THEOREM A.4. *Every discriminating algorithm is bounded-exploration.*

PROOF . Let A be a discriminating algorithm. We will obtain a bounded-exploration witness T for A. We will use

—the compactness theorem for first-order logic found in standard textbooks on mathematical logic, and

—the fact, proved in standard textbooks on topology, that the Cantor space is compact.

To recall, the Cantor space consists of infinite sequences $\xi = \langle \xi_i : i \in N \rangle$ where N is the set $\{0, 1, \ldots\}$ of natural numbers and each $\xi_i \in \{0, 1\}$. Every function f from a finite set of natural numbers to $\{0, 1\}$ gives a basic open set

$$O(f) \rightleftharpoons \{\xi : \xi_i = f(i) \text{ for all } i \in \text{Domain}(f)\}$$

of the Cantor space. An arbitrary open set of the Cantor space is the union of a collection of basic open sets $O(f)$.

Let Υ be the vocabulary of A, and let $\mathcal{T}$ be a discrimination witness for A. In this proof, terms and structures are of vocabulary Υ, states are those of A, pseudo-states are pseudo-states of A with respect to $\mathcal{T}$, and error states are error states of A with respect to $\mathcal{T}$.

LEMMA A.5. *Suppose that X, Y are legal states and Z is a pseudo-state. If Z coincides with X over $\mathcal{T}(X)$ and with Y over $\mathcal{T}(Y)$, then $\mathcal{T}(X) = \mathcal{T}(Y)$.*

PROOF . All three structures coincide over $\mathcal{T}(X) \cap \mathcal{T}(Y)$. By the second condition of Definition A.1, $\mathcal{T}(X) = \mathcal{T}(Y)$. □

Definition A.2 imposes no restrictions on the sets $\mathcal{T}(Y)$ where Y ranges over pseudo-states. Due to Lemma A.5, we may assume this: if a pseudo-state Y coincides with a legal state X over $\mathcal{T}(X)$ then $\mathcal{T}(Y) = \mathcal{T}(X)$.

COROLLARY A.6. *If a legal state X coincides with a structure Y over $\mathcal{T}(Y)$, then $\mathcal{T}(X) = \mathcal{T}(Y)$.*

PROOF . By the second requirement of Definition A.2, Y cannot be an error state. We consider the two remaining cases.

Case 1: Y is a legal state. Use the second requirement of Definition A.1.

Case 2: Y is a pseudo-state. Then there exists a legal state Z such that Y coincides with Z over $\mathcal{T}(Z)$ and therefore $\mathcal{T}(Y) = \mathcal{T}(Z)$. Hence X coincides with Z over $\mathcal{T}(Z)$. Then, as in Case 1, $\mathcal{T}(X) = \mathcal{T}(Z)$. Hence $\mathcal{T}(X) = \mathcal{T}(Y)$. □

Call a term *equational* if it has the form $t_1 = t_2$. List all equational terms in some order:

$$e_1, e_2, e_3, \ldots.$$

Each structure X gives rise to a binary sequence χ^X where $\chi^X_i = 1$ if e_i holds in X, and $\chi^X_i = 0$ if e_i fails in X.

Let $O(X)$ be the basic open subset of the Cantor space that consists of sequences ξ with $\xi_i = \chi^X_i$ for every equation e_i such that both sides of e_i belong to $\mathcal{T}(X)$.

LEMMA A.7. *Let ξ be a binary sequence that does not belong to any $O(X)$. There exists an open set $O(\xi)$ that contains ξ and is disjoint from every $O(X)$.*

PROOF . Consider a first-order system of axioms in vocabulary Υ which contains the usual equality axioms and includes the infinite set

$$D \rightleftharpoons \{\varphi_i : i = 1, 2, 3, \ldots\}$$

where φ_i is e_i if $\xi_i = 1$, and φ_i is $\neg e_i$ if $\xi_i = 0$.

If every finite subset of D is satisfiable, then — by the compactness theorem for first-order logic — D is satisfiable and therefore has a model X. But then $\xi \in O(X)$ which contradicts our assumption about ξ.

Hence there is a finite unsatisfiable fragment D_0 of D. The desired $O(\xi)$ consists of all sequences ξ' such that $\xi'_i = \xi_i$ for all i with $\varphi_i \in D_0$. □

The open sets $O(X)$ and $O(\xi)$ cover the Cantor space. Since Cantor space is compact, it is covered by a finite collection of these open sets. Therefore there is a finite collection $\{X_1, \ldots, X_k\}$ of states such that

$$\{\chi^X : X \text{ is a state}\} \subseteq O(X_1) \cup \cdots \cup O(X_k).$$

The set $T = \mathcal{T}(X_1) \cup \cdots \cup \mathcal{T}(X_k)$ witnesses that A is bounded-exploration. Indeed, let states Y and Z coincide over T. There exists $i \in \{1, \ldots, k\}$ such that $Y \in O(X_i)$, so that Y coincides with X_i over $\mathcal{T}(X_i)$. By Corollary A.6, $\mathcal{T}(Y) = \mathcal{T}(X_i)$, so that $\mathcal{T}(Y) \subseteq T$. It follows that Z coincides with Y over $\mathcal{T}(Y)$. By the first requirement of Definition A.1, $\Delta(A, Y) = \Delta(A, Z)$. That completes the proof of the theorem. □

Let A be a finite-exploration algorithm with a finite exploration witness $\mathcal{T}$, and let Υ be the vocabulary of A. As in the proof of Theorem A.4, terms and structures are of vocabulary Υ, states are those of A, and pseudo-states (respectively error states) are pseudo-states (respectively error states) of A with respect to $\mathcal{T}$.

Call two structures *similar* if no equational term distinguishes between them: if an equation holds in one of the structures then it holds in the other. For every structure X, let $[X]$ be the collection of all structures Y similar to X, the *similarity class* of X.

LEMMA A.8. *Every structure similar to an error state is an error state.*

PROOF . Suppose that a structure Z is similar to an error state Y. If Z is a legal state or a pseudo-state, then there is a legal state X such that Z coincides with X over $\mathcal{T}(X)$. Since Y is similar to Z, Y also coincides with X over $\mathcal{T}(X)$, so that Y is a pseudo-state, which contradicts our assumption that Y is an error state. □

Let the mapping $X \mapsto \chi^X$ be as in the proof of Theorem A.4. The mapping $[X] \mapsto \chi^X$ is one-to-one (but not onto). Use it to define a topological space TS of the similarity classes of structures; the open sets of TS are the pre-images of the open sets of the Cantor space.

THEOREM A.9. *If there is a closed set F in TS such that*

—F contains all the similarity classes $[X]$ where X is a legal state, and

—F contains no similarity classes $[Y]$ where Y is an error state

then A is bounded-exploration.

PROOF . Let Y be an error state. The complement of F in TS is an open set that contains $[Y]$ but does not contain any $[X]$ where $[X]$ is a legal state. By the definition of TS, there is an open subset of the Cantor space that contains χ^Y but does not contain any χ^X where X is a legal state. Hence there is a basic open set $O(f)$ that contains χ^Y but does not contain any χ^X where X is a legal state. Define $\mathcal{T}(Y)$ to be the set of all subterms of equational terms e_i with $i \in \text{Domain}(f)$. No legal state X of A coincides with Y over $\mathcal{T}(Y)$; otherwise χ^X would belong to $O(f)$ which is impossible.

The extended map $\mathcal{T}$ is a discrimination witness for A. By Theorem A.4, A is bounded-exploration. □

REFERENCES

ASM MICHIGAN WEBPAGE. `http://www.eecs.umich.edu/gasm/`, maintained by J. K. Huggins.

BLASS, A. AND GUREVICH, Y. 1997. The linear time hierarchy theorem for RAMs and abstract state machines. *Springer Journal of Universal Computer Science 3*, 4, 247–278.

BLASS, A., GUREVICH, Y., AND SHELAH, S. 1999. Choiceless polynomial time. *Annals of Pure and Applied Logic 100*, 141–187.

BLUM, L., SHUB, M., AND SMALE, S. 1989. On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bulletin of American Mathematical Society 21*, 1–46.

BÖRGER, E. 1995. Why use evolving algebras for hardware and software engineering? *Springer Lecture Notes in Computer Science 1012*, 236–271.

BÖRGER, E. 1999. High level system design and analysis using abstract state machines. *Springer Lecture Notes in Computer Science 1641*, 1–43.

BÖRGER, E., GRÄDEL, E., AND GUREVICH, Y. 1996. *Classical Decision Problem*. Springer.

BÖRGER, E. AND HUGGINS, J. K. 1998. Abstract state machines 1988–1998: Commented ASM bibliography. *Bulletin of European Association for Theoretical Computer Science*. Number 64, February, pp. 105–128.

CHANDY, K. M. AND MISRA, J. 1988. *Parallel Program Design*. Addison-Wesley.

CHURCH, A. 1936. An unsolvable problem of elementary number theory. *American Journal of Mathematics 58*, 345–363. Reprinted in [Davis 1965, 88–107].

COOK, S. A. AND RECKHOW, R. A. 1973. Time-bounded random access machines. *Journal of Computer and Systems Sciences 7*, 354–475.

DAVIS, M. 1965. *The Undecidable*. Raven Press.

Gandy, R. 1980. Church's thesis and principles for mechanisms. In J. Barwise, H. J. Keisler, and K. Kunen Eds., *The Kleene Symposium*, pp. 123–148. North-Holland.

Gries, D. 1990. The maximum-segment-sum problem. In E. W. Dijkstra Ed., *Formal Development of Programs and Proofs*, pp. 33–36. Addison-Wesley.

Grigoriev, D. 1976. Kolmogorov algorithms are stronger than Turing machines. In Y. Matiyasevich and A. Slisenko Eds., *Investigations on Constructive Mathematics and Math. Logic VII*, pp. 29–37. Nauka. In Russian.

Gurevich, Y. 1985. A new thesis. *American Mathematical Society Abstracts*. August 1985, p. 317.

Gurevich, Y. 1988. Kolmogorov machines and related issues. In G. Rozenberg and A. Salomaa Eds., *Current Trends in Theoretical Computer Science*. World Scientific, 1993, pp. 225–234. A reprint of the article in the Bulletin of European Association for Theoretical Computer Science, Number 35 (1988), pp. 71–82.

Gurevich, Y. 1991. Evolving algebras: An attempt to discover semantics. In G. Rozenberg and A. Salomaa Eds., *Current Trends in Theoretical Computer Science*. World Scientific, 1993, pp. 266–292. A slight revision of the tutorial in the Bulletin of European Association for Theoretical Computer Science, Number 43 (1991), pp. 264–284.

Gurevich, Y. 1995. Evolving algebra 1993: Lipari guide. In E. Börger Ed., *Specification and Validation Methods*, pp. 9–36. Oxford University Press.

Gurevich, Y. 1997. May 1997 draft of the ASM guide. University of Michigan EECS Department Technical Report CSE-TR-336-97 found at [ASM Michigan Webpage].

Gurevich, Y. 1999. The sequential ASM thesis. *Bulletin of European Association for Theoretical Computer Science*. Number 67, February 1999, pp. 93–124.

Gurevich, Y. 2000. Puget Sound ASM guide. In preparation, the title is tentative.

Gurevich, Y. and Spielmann, M. 1997. Recursive abstract state machines. *Springer Journal of Universal Computer Science 3*, 4, 233–246.

Knuth, D. E. 1968. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley.

Kolmogorov, A. N. 1953. On the notion of algorithm. *Uspekhi Mat. Nauk 8*, 175–176. in Russian.

Kolmogorov, A. N. and Uspensky, V. A. 1958. On the definition of algorithm. *Uspekhi Mat. Nauk 13*, 3–28. in Russian. English translation in AMS Translations 29 (1963), 217–245. See also [Kolmogorov 1953].

Markov, A. A. 1954. Theory of algorithms. *Transactions of the Steklov Institute of Mathematics 42*. In Russian. An English translation published by the Israel Program for Scientific Translations, Jerusalem, 1962.

Savage, J. E. 1987. *The Complexity of Computing*. Robert E. Krieger, Malabar, Florida.

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

Schönhage, A. 1970. Universelle Turing Speicherung. In Dörr and Hotz Eds., *Automatentheorie und Formale Sprachen*, pp. 369–383. Bibliogr. Institut, Mannheim. In German.

Schönhage, A. 1980. Storage modification machines. *SIAM Journal on Computing 9*, 490–508. See also [Schönhage 1970].

Tarski, A. 1933. The concept of truth in the languages of deductive sciences. *Prace Towarzystwa Naukowego Warszawskiego III, no. 34*. In Polish. English translation in J. H. Woodger Ed., *Logic, Semantics, Metamathematics: Papers from 1923 to 1938*, Clarendon Press, Oxford, 1956, pp. 152–278.

Turing, A. 1936. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of London Mathematical Society (2) 42 (1936–37)*, 230–236. Correction, *ibid.* 43, 544–546. Reprinted in [Davis 1965, 155–222].

Uspensky, V. A. 1992. Kolmogorov and mathematical logic. *Journal of Symbolic Logic 57*, 385–412.