# Exact Exploration and Hanging Algorithms[⋆]

Andreas Blass,[1] Nachum Dershowitz[2] and Yuri Gurevich[3]

[1] Mathematics Dept, University of Michigan, Ann Arbor, MI 48109, USA
[2] School of Computer Science, Tel Aviv University, Ramat Aviv 69978, Israel
[3] Microsoft Research, Redmond, WA 98052, USA

**Abstract.** Recent analysis of sequential algorithms resulted in their axiomatization and in a representation theorem stating that, for any sequential algorithm, there is an abstract state machine (ASM) with the same states, initial states and state transitions. That analysis, however, abstracted from details of intra-step computation, and the ASM, produced in the proof of the representation theorem, may and often does explore parts of the state unexplored by the algorithm. We refine the analysis, the axiomatization and the representation theorem. Emulating a step of the given algorithm, the ASM, produced in the proof of the new representation theorem, explores exactly the part of the state explored by the algorithm. That frugality pays off when state exploration is costly. The algorithm may be a high-level specification, and a simple function call on the abstraction level of the algorithm may hide expensive interaction with the environment. Furthermore, the original analysis presumed that state functions are total. Now we allow state functions, including equality, to be partial so that a function call may cause the algorithm as well as the ASM to hang. Since the emulating ASM does not make any superfluous function calls, it hangs only if the algorithm does.

*[T]he monotony of equality can only lead us to boredom.*

*—Francis Picabia*

## 1 Introduction

According to Kolmogorov, “algorithms compute in steps of bounded complexity” [14]. We call such algorithms sequential; in the intervening years the notion of algorithm was generalized to computations that may be vastly parallel, distributed, real-time. In the rest of this paper, algorithms are by default sequential and deterministic. In particular abstract state machines [12] will be by default sequential and deterministic.

Abstract state machines (ASMs) constitute a most general model of (sequential deterministic) computation. They operate on any level of abstraction of data structures and native operations. By virtue of the ASM Representaion

[⋆] Blass was partially supported by NSF grant DMS-0653696. Dershowitz was partially supported by Israel Science Foundation grant 250/05. Part of the work reported here was performed during visits of the first two authors to Microsoft.

Theorem of [13], any algorithm can be step-by-step emulated by an ASM. The theorem presupposes a precise notion of algorithm, and indeed algorithms are axiomatized by means of three "sequential postulates" in [13]. These postulates formalize the following intuitions:

**(I)** an algorithm is a state-transition system;
**(II)** state information determines (given the program of the algorithm) future transitions and may be captured by a logical structure;
**(III)** state transitions are governed by the values of a finite and input-independent set of ground terms.

All models of effective, sequential computation satisfy the postulates, as do idealized algorithms for computing with real numbers, or for geometric constructions with compass and straightedge. Careful analysis of the notion of algorithm [13] and an examination of the intent of the founders of the field of computability [10] have demonstrated that the sequential postulates are true of all sequential, deterministic algorithms, the only kind envisioned by the pioneers of the field. In Sects. 3 and 4, we explain the postulates, recall ASMs and formulate the ASM representation theorem.

The algorithms of the three sequential postulates will be called classical. The notion of behavioral equivalence for classical algorithms is rather strict: behaviorally equivalent algorithms have the same states, the same initial states and the same state transitions. The ASM representation theorem asserts that, for every classical algorithm, there is a behaviorally equivalent ASM. In various application domains, weaker notions of equivalence — e.g. bisimulation — may be useful, but the representation theorem remains valid for any weakening of the notion of behavioral equivalence.

Yet, from a certain point of view, the notion of behavioral equivalence is not strict enough. Equivalent algorithms may have different intra-state behavior. In particular the emulating ASM produced in the proof of the ASM representation theorem may and usually does explore parts of the state unexplored by the given algorithm. Superfluous evaluations do not prevent the ASM from arriving at the same transition as the algorithm it emulates but they waste resources. For example, an algorithm for removing duplicates from a file system may test equality of large files, but would first check to see that their recorded sizes are the same. The ASM produced by the proof of the ASM representation theorem would, however, always check both size and content, despite the overhead.

The universal construction of the emulating ASM was designed to simplify the proof of the ASM representation theorem. That construction was not designed to be used in applications. In fact, by the time of the publication of the ASM representation theorem, the ASM community had developed an art of efficient — and elegant — ASM emulation and had accumulated substantial evidence that efficient emulation of intra-step computations was always possible.

In the present paper, we prove that efficient emulation of intra-step computations is indeed always possible. In Sect. 5, we refine the axiomatization of algorithms. The algorithms of the new postulates are called exacting. And we strengthen the notion of behavioral equivalence. Two exacting algorithms are behaviorally equivalent if

**(i)** they have the same states, initial states and state transitions and
**(ii)** at each step they explore the same part of the state.

The new ASM representation theorem, in Sect. 7, asserts that, for every exacting algorithm, there is a behaviorally equivalent ASM. By eliminating unnecessary exploration, the emulating ASM, produced by the proof of the new representation theorem, is often simpler and shorter — with no need for human ingenuity to improve it.

Exact exploration allows us to handle faithfully algorithms that may hang. To this end, we liberalize the notion of algorithm's state. States of classical algorithms are (first-order) structures except that relations are viewed as Boolean-valued functions. Basic functions of the state (i.e. the interpretations of the function names in the vocabulary) are total. In the case of exacting algorithms, basic functions may be partial. That may sound like old news to ASM experts. Even though basic functions of classical ASMs are supposed to be total, partial functions are easily handled by means of various "error values", the most popular of which has been undef. The error values are elements of the state. For example, in a state with integer arithmetic, you may have that term $1/0$ evaluates to undef, equality term $1/0 = 7$ evaluates to false, and equality term $1/0 = \mathsf{undef}$ evaluates to true. Here we are talking about a very different situation. A basic function $f$ may have no value whatsoever at some tuple $\bar{a}$ of arguments. When $f(\bar{a})$ is called, the algorithm hangs (or stalls). This is very different from returning an error value. Hanging is more insidious – the computation gets stuck in a catatonic limbo, while an error value allows the algorithm to handle the situation as it sees fit.

Even equality can be partial in the states of an exacting algorithm. Why that? Consider the following scenario. An algorithm works with genuine (infinite precision) real numbers. Internally, real numbers have finite representations, e.g. definite integrals. The problem when two such expressions represent the same real number is undecidable of course. Accordingly, in algorithm's states, equality is partial. It need not be even transitive. It could be that a test $s = t$ yields false, whereas $t = u$ yields true, yet when the state asked about $s = u$, no answer is forthcoming, though the truth of the matter must be that $s \neq u$.

The possibility of hanging makes exact exploration crucial. Consider for example the Gaussian elimination procedure. It tests that a pivot element $p$ is non-zero before evaluating expressions $a[i,j]/p$. In the case $p = 0$, it does not evaluate expressions $a[i,j]/p$ but the ASM, produced by the proof of the classical ASM representation theorem, does. In contrast, the ASM, produced by the proof of the new ASM representation theorem in Sect. 7, does not conduct such superfluous evaluations.

**Acknowledgment.** We thank Olivier Bournez for his comments on an early draft and Ulrich Kohlenbach for information on computable reals.

## 2 Related Work

Exact exploration that we preach here has been practiced by ASM experts for a long time, in various applications in academia and industry [1,7,25]. ASMs, sequential and otherwise, have been used to give high-level operational semantics to programming languages, protocol specifications, etc. In Microsoft, the ASM approach was used to develop Spec Explorer, a top tool for model-based testing.

The axiomatization of algorithms in [13], which is extended here to account for exact exploration, was extended to parallel algorithms and to interactive algorithms [6].

We are aiming for a model of computation that can faithfully support algorithms for which basic operations may have varying costs involved, and/or for which their domains of applicability may be unknown or uncomputable. The latter produces an ocean of related work.

First, constructive mathematics comes to mind [3,16]. Classical mathematics has no philosophical objections to working with various ideal elements that do not have finite representations. The BSS model of computation with real numbers reflects that attitude [4], and ASMs have been used in to emulate the BSS model [24]. On the other hand, constructive mathematics works only with objects that have finite representations. In their world, only computable reals exist, only computable reals-to-reals functions exist, etc. The question when two computable reals are equal is of course undecidable. Would ASMs be of any use to constructivists? We think so. Russian constructivists often used Markov's normal algorithms computation model [18] for programming. As a result some of their works are unnecessarily detailed and hard to read. Exacting ASMs would fit their purposes better.

You don't have to be a constructivist to be interested in computable mathematical analysis; you may be a recursion theorist [28]. One way to deal with partial-equality troubles is to avoid equality altogether [15,23]. Algebraic semantics has been used to tackle partial-functions difficulties in abstract data types and programming semantics; see [2,19] for interesting examples of that approach.

There exist a number of implementations of arithmetic with infinite-precision reals; see [11] for a survey. As far as we can determine, the most advanced and rapid implementations of exact real number arithmetic today are the iRRAM system of Norbert Müller [22] and the RealLib system by Branimir Lambov [17]. Let us mention also the xrc system (alluding to Exact Reals in C) of Keith Briggs [9] and a Common Lisp package Computable Real Numbers by Michael Stoll [26]. But there are other systems of interest.

One of our reviewers suggested that we "ought to engage with" Winskel's event structures [29] and noted a similarity between our Discrimination requirement in Sect. 5 and Winskel's coincidence-freeness of configurations. Well, the particular structure of exact exploration can be made to concur with a number of general frameworks including that of event structures. One may think of the exploration ordering as a partial ordering of term-evaluation events connected to an event structure associated to a given algorithm at a given state. And indeed there is some resemblance between Discrimination and coincidence-freeness.

Discrimination asserts the existence of a partial order reflecting an intuition of causality, and coincidence-freeness can be cast in similar terms. But the resemblance is rather superficial; in most cases, the actual partial orderings differ.

The same reviewer also asked how our framework is related to Moschovakis's "abstract computation theories". Moschovakis has defined an abstract notion of recursor and has proposed that algorithms be identified with recursors; see for example [20,21]. For a discussion of this proposal in the light of ASMs, see [5]. We concentrate here on the aspect of Moschovakis's proposal connected with the main issue of the present paper, namely "What does an algorithm actually look at?" This issue arises implicitly in [20], but in a quite different context. Instead of being defined directly from explicit instructions in the algorithm, a set, which intuitively contains what the algorithm looks at, is obtained as the conclusion of a theorem for a specific example (see [20, Theorem 5.2]. The context there is estimating an algorithm's usage of a certain resource by checking how much of that resource it looks at. An important difference from our discussion is that in [20] the "looking" refers to an entire run of the algorithm, whereas our $\Gamma(X)$ refers to a single step. The latter can often be read off from a program (for example if the program is an ASM); the former, on the other hand, can generally be found only by running the program. Although the cited theorem in [20] provides an appropriate notion of "looking at" for the specific algorithm (mergesort) considered there, algorithms in general will not admit such explicit bounds on what they look at.

## 3 Axiomatization of Algorithms

Algorithms were axiomatized in [13]. Here we describe a slightly refined axiomatization that allows for a partial transition function.

### 3.1 Sequential Time

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

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

1. A nonempty set[4] $\mathcal{S}$ of *states* and a nonempty subset $\mathcal{I} \subseteq \mathcal{S}$ of *initial states.*
2. A partial *next state* transition function $\tau$ from $\mathcal{S}$ to $\mathcal{S}$.

Having $\tau$ depend only on the state means that states must store all the information needed to determine subsequent behavior. Prior history is unavailable unless stored in the current state. If $\tau(X)$ is undefined, we say that $X$ is *terminal* and write $\tau(X) = \bot$.

[4] Or class; the distinction is irrelevant for our purposes and we shall ignore it.

### 3.2 Abstract State

Our notion of structure is that of first-order logic with equality except for the following modifications that are inessential but convenient for our purposes (and standard in the ASM literature).

- Propositional constants true and false are viewed as elements of the structure (and thus "live" inside the structure rather than outside).
- Relations are viewed as Boolean-valued functions.
- Standard Boolean connectives $\neg$, $\wedge$, $\vee$ are viewed as structures's basic functions. (The basic functions of a structure are the interpretations of the vocabulary's function names.)

The Boolean values, equality and the Boolean connectives are the *logic basic functions* of any structure; their names form the logic part of the structure's vocabulary, and they are interpreted as expected. As usual, constants are nullary functions.

All basic functions are total. A basic function may return an "error value", e.g. the value undef mentioned earlier, but error values denote elements of the structure. We write $\mathrm{Dom}\, X$ for the base set of a structure $X$.

**Postulate II (Abstract State).**

The states of an algorithm are structures over a finite vocabulary $\mathcal{F}$ such that the following conditions are satisfied.

1. If $X$ is a state of the algorithm, then any structure $Y$ isomorphic to $X$ is also a state, and $Y$ is initial or terminal if $X$ is initial or terminal, respectively.
2. Transition $\tau$ preserves the base set; i.e. $\mathrm{Dom}\, \tau(X) = \mathrm{Dom}\, X$ for every non-terminal state $X$.
3. Transition $\tau$ respects isomorphisms, so, if $\zeta : X \cong Y$ is an isomorphism of non-terminal states $X, Y$, then $\zeta : \tau(X) \cong \tau(Y)$.

Closure under isomorphism ensures that the algorithm operates on the chosen level of abstraction; the states' internal representation of the data is invisible and immaterial to the program. Since a state $X$ is a structure, it interprets function symbols in $\mathcal{F}$, assigning a value $b$ from $\mathrm{Dom}\, X$ to the "location" $f(a_1, \ldots, a_k)$ in $X$ for every $k$-ary symbol $f \in \mathcal{F}$ and for every tuple $a_1, \ldots, a_k$ in $\mathrm{Dom}\, X$. In this way, $X$ assigns a value $[\![t]\!]_X$ in $\mathrm{Dom}\, X$ to terms $t$ over $\mathcal{F}$.

It is convenient to view each state as the union of the graphs of its operations, given in the form of a set of location-value pairs, each written conventionally as $f(\bar{a}) \mapsto b$. Define the *update set* $\Delta(X)$ of state $X$ as the set $\tau(X) \setminus X$ of changed pairs where $\Delta(X) = \bot$ if $\tau(X) = \bot$. $\Delta$ encapsulates the state-transition function $\tau$ by providing all the information necessary to update the current state. But to produce $\Delta(X)$, the algorithm needs to evaluate, with the help of the information stored in $X$, the values of some terms. Later, we will use $\Gamma(X)$ to refer to the set of these "exploration" terms.

### 3.3 Bounded Exploration

The third postulate simply states that there is a fixed, finite set of ground (variable-free) terms that determines the behavior of the algorithm. We say that states $X$ and $Y$ *agree* on a set $T$ of terms, and we write $X =_T Y$, if $[\![t]\!]_X = [\![t]\!]_Y$ for all $t \in T$.

**Postulate III (Bounded Exploration).** For every algorithm, there is a finite set $T$ of ground *critical terms* over the state vocabulary such that, for all states $X$ and $Y$, if $X =_T Y$ then $\Delta(X) = \Delta(Y)$.

In what follows, we will presume that the set $T$ of critical terms contains $\mathsf{true}$, $\mathsf{false}$ and is closed under subterms. Algorithms satisfying Postulates I, II, and III will be called *classical*. It is argued in [13] that every (sequential deterministic) algorithm is classical in that sense.

If $\Delta(X) = \Delta(Y)$ and one of the two states is terminal then so is the other. It follows that states $X$ and $Y$ of a classical algorithm have the following property: if they agree on all critical terms then either both of them are terminal or else neither is terminal and the update sets $\Delta(X)$ and $\Delta(Y)$ coincide.

## 4 Abstract State Machines

### 4.1 ASM Programs

An *ASM program* $P$ over a vocabulary $\mathcal{F}$ takes one of the following forms:

- an *assignment* statement $f(s_1, \ldots, s_n) := t$, where $f \in \mathcal{F}$ is a function symbol of arity $n$, $n \geq 0$, and $s_i$ and $t$ are ground terms over $\mathcal{F}$;
- a *parallel* statement $P_1 \parallel \cdots \parallel P_n$ ($n \geq 0$), where each $P_i$ is an ASM 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 $\mathcal{F}$.

*Example 1.* Here is a sorting program:

```
if $j \neq n$ then
    if $F(i) > F(j)$ then $F(i) := F(j) \parallel F(j) := F(i)$
    $j := j + 1$
if $j = n \wedge i + 1 \neq n$ then $i := i + 1 \parallel j := i + 2$
```

where, as the indentation hints, the two statements given by lines 2 and 3 respectively are two components of a parallel combination, and similarly the two statements given by lines 1–3 and by line 4 respectively form a parallel combination. And $j \neq n$ is short for $\neg(j = n)$. The extra-logic part of program's vocabulary is $\{1, 2, +, >, F, n, i, j, \mathsf{undef}\}$.

Every state of the sorting program interprets the symbols $1, 2, +, >$ as usual. These are static; their interpretation will never be changed by the program. The semantics of statements is as expected. The program, as such, defines a single step, which is repeated forever or until it arrives to a terminal state. Fixed nullary functions 0 and $n$ (programming "constants") serve as bounds of an array $F$, where $F$ is a unary function. In addition, varying nullary functions $i$ and $j$ (programming "variables") are used as array indices. Initial states have $n \geq 0$, $i = 0$, $j = 1$, integer values for $F(0), \ldots, F(n-1)$, and $\mathsf{undef}$ for all other points of $F$. The algorithm proceeds by modifying the values of $i$ and $j$ as well as of locations $F(0), \ldots, F(n-1)$. It always terminates successfully with $j = n = i + 1$ and with the first $n$ elements of $F$ sorted.

All terms in the sorting program are critical, and no other critical terms are needed. In general, the left-hand sides of assignments contribute only proper subterms to the set of critical terms but in this example the left sides are critical since they occur also elsewhere in the program.

### 4.2 ASM Updates

An ASM might "update" a location in a *trivial* way, giving it the 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 ASM fails. To take these additional possibilities into account, a *proposed* update set $\Delta^+_P(X)$ for an ASM $P$ at state $X$ is defined as follows:

- $\Delta^+_{f(\ldots,s_i,\ldots):=t}(X) = \{f(\ldots, [\![s_i]\!]_X, \ldots) \mapsto [\![t]\!]_X\}$;
- $\Delta^+_{[P_1 \,\|\, \cdots \,\|\, P_n]}(X) = \begin{cases} \Delta^+_{P_1}(X) \cup \cdots \cup \Delta^+_{P_n}(X) & \text{if there is no clash} \\ \bot & \text{otherwise;} \end{cases}$
- $\Delta^+_{\mathbf{if}\, C\, \mathbf{then}\, P}(X) = \Delta^+_P(X)$ if $X \models C$, and $\varnothing$ otherwise.

Here $X \models C$ means of course that $C$ evaluates to $\mathsf{true}$ in $X$, and we stipulate that the union of $\bot$ with any set is $\bot$. If $\Delta^+_P(X) = \bot$ or $\Delta^+_P(X) = \varnothing$, $X$ is a terminal state of $P$. 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 $\Delta^+_P(X)$. So, if the latter contains only trivial updates, $P$ will loop forever. (As long as no confusion will arise, we are dropping the subscript $P$.)

ASMs clearly satisfy Postulates I–III, and thus the notion of update set $\Delta(X)$, defined after Postulate II, applies to any ASM $P$. Let $\Delta^0(X)$ denote the set $\{f(\bar{a}) \mapsto [\![f(\bar{a})]\!]_X \mid \bar{a} \in \mathrm{Dom}\, X\}$ of all possible trivial updates for state $X$. Thus $X$ is terminal if $\Delta^+_P(X)$ is $\varnothing$ or $\bot$, and $\Delta_P(X) = \Delta^+_P(X) \setminus \Delta^0(X)$ otherwise. The update set for the sorting program, when in a state $X$ such that $[\![j]\!]_X \neq [\![n]\!]_X$ and $[\![F(i)]\!]_X > [\![F(j)]\!]_X$, contains $F([\![i]\!]_X) \mapsto [\![F(j)]\!]_X$, $F([\![j]\!]_X) \mapsto [\![F(i)]\!]_X$, $j \mapsto [\![j]\!]_X + 1$.

### 4.3 Classical ASM Representation Theorem

Two classical algorithms are *behaviorally equivalent* if they have the same states, the same initial states and the same state transitions.

**Theorem 1.** *Every classical algorithm has a behaviorally equivalent ASM.*

The proof constructs an ASM that contains conditions involving equalities and disequalities between all critical terms. These conditions can be large. Given the critical terms for our sort algorithm, the ASM constructed by the proof in [13] would include statements like

$$\textbf{if}\ (F(i) > F(j)) = \mathsf{true} \wedge j = n \wedge i+1 \neq n\ \textbf{then}\ j := i+2.$$

This, despite the fact that the first conjunct of the conditional is irrelevant when the others hold.

## 5 Refined Axiomatization of Algorithms

The sequential-time Postulate I remains unchanged. We liberalize the abstract-state Postulate II to a new abstract-state Postulate II-lib, and replace the bounded-exploration Postulate III with an exact-exploration Postulate III-exact.

### 5.1 Liberalization of Abstract State Postulate

It is not uncommon in the logic literature to generalize the notion of structure so that basic non-logic functions may be partial. We do that. But we also do something that is not common: we allow equality to be partial. The reason for that was mentioned in the Introduction. Recall that scenario where an algorithm works with real numbers represented internally by expressions like definite integrals. In that scenario, the state may not know whether two reals are equal, i.e. whether their representations denote the same real number; the state may not know whether a real number equals zero. We insist, however, that true and false are defined (and thus total), and that the Boolean connectives are total; there is no reason to make these logic functions partial. And yes, equality remains true equality whenever it is defined. Thus equality is semi-logical: partial but correct when defined.

**Postulate II-lib (Abstract State).** The states of an algorithm are structures over a finite vocabulary $\mathcal{F}$, where equality and non-logic basic functions may be partial, such that the following conditions are satisfied.

1. If $X$ is a state of the algorithm, then any structure $Y$ isomorphic to $X$ is also a state, and $Y$ is initial or terminal if $X$ is initial or terminal, respectively.
2. Transition $\tau$ preserves the base set; i.e. $\operatorname{Dom}\tau(X) = \operatorname{Dom}X$ for every non-terminal state $X$. And $\tau$ cannot change a value at any location $f(\bar{a})$ to no value.
3. Transition $\tau$ respects isomorphisms, so, if $\zeta : X \cong Y$ is an isomorphism of non-terminal states $X, Y$, then $\zeta : \tau(X) \cong \tau(Y)$.

Conditions 1 and 3 are exactly as in Postulate II. Only condition 2 is amended with a restriction on $\tau$.

Thus some locations $f(\bar{a})$ may have no value whatsoever, not even an error value like undef; in such cases we write $f(\bar{a}) = \bot$. If an algorithm attempts, in some state $X$, to access a non-existent value, then it must hang. There will be no next state, yet the algorithm will get no indication of this failure. (Any such indication would be just an error value; as we said, error values can treated as state elements.) If this situation occurs at any point during the evaluation of a term $t$, then $t$ has no value in state $X$, symbolically $[\![t]\!]_X = \bot$. Thus $[\![t]\!]_X = \bot$ if $[\![t']\!]_X = \bot$ for any subterm $t'$ of $t$.

Define the domain $\mathrm{Dom}\, f$ of a function $f$ in state $X$ to be the set of all tuples $\bar{a}$ such that the location $f(\bar{a})$ has a value in $X$, possibly an error value like $\mathsf{undef}$. The restriction that we impose on the transition function $\tau$ says that $\mathrm{Dom}\, f$ can only grow.

### 5.2 Exact Exploration Postulate

Deciding which locations to explore is now part of the behavior we are interested in. If an algorithm acts differently on different states, either in the sense of exploring different terms or in the sense of performing different updates, then it must *first* find some term that distinguishes them. Furthermore, if the behaviors of the algorithm in two states differ, then that must be made evident from that part of the two states that is explored in both. Accordingly, what we should have is

$$X =_{\Gamma(X)\cap\Gamma(Y)} Y \longrightarrow \Gamma(X) = \Gamma(Y) \tag{1}$$

$$X =_{\Gamma(X)\cap\Gamma(Y)} Y \longrightarrow \Delta(X) = \Delta(Y) \tag{2}$$

In order to compute over a state $X$, the program evaluates – in some order – finitely many Boolean terms over $X$ and learns their values. In order to produce updates, additional terms may be evaluated but the program does not need to know their values. The state may not determine the precise order of exploration, but some partial order is dictated by the possible behaviors. In general, if a conditional statement **if** $C$ **then** $P$ is executed and the test $C$ is true, then the terms in $C$ are explored before, or together with, those in $P$. One cannot, however, simply derive the exploration order from the conditionals in the program, making conditions in $C$ precede any new terms in $P$. We might have an assignment **if** $d$ **then if** $b$ **then** $x := d$, in which case $d$ needs to be explored before $b$, but when this assignment is placed in parallel with **if** $b$ **then if** $d$ **then** $x := c$, $b$ and $d$ can be explored at the same time. So, instead, we put all terms of the top-level conditions and assignments of components of a parallel statement at the bottom of the ordering, followed by contributions from the relevant cases of the conditionals. This order of exploration will be captured in what follows by a "causality" order $\prec_X$ on the explore terms $\Gamma(X)$ of states $X$. For example, the order for the sorting program, when $[\![j]\!]_X \neq [\![n]\!]_X$ and $[\![F(i)]\!]_X > [\![F(j)]\!]_X$ has the two conditions $j \neq n$ and $j = n \wedge i+1 \neq n$ incomparable, with the first of these conditions being below both $F(i) > F(j)$ and $j+1$.

*Example 2.* Consider this parallel combination of three ASM statements:

```
if $d$ then if $c$ then if $b$ then $s := x$
if $d$ then if $\neg c$ then $t := x$
if $d$ then if $\neg b$ then $s := y$ .
```

Clearly, $d$ must be explored first off, since nothing more transpires when $\neg d$, while further tests are necessary when $d$ holds, in which case, $b$ and $c$ must both be explored, though the order in which that occurs does not matter. Of course, $x$ and/or $y$ are only explored after it becomes clear that the relevant case holds.

**Postulate III-exact (Exact Exploration).** For every algorithm, there is a finite set $T$ of ground *critical terms* over the state vocabulary such the following holds. For every state $X$, there is a finite *explore set* $\Gamma(X) \subseteq T$ satisfying the following two properties.

1. **Determination.** For every state $Y =_{\Gamma(X)} X$, $\Delta(Y) = \Delta(X)$.
2. **Discrimination.** There is a partial order $\prec_X$ of $\Gamma(X)$ such that, for every state $Y$ and every $t \in \Gamma(X) \setminus \Gamma(Y)$, there is a Boolean term $s \prec_X t$ that takes on opposite truth values in $X$ and $Y$.

The intention is that the explore set $\Gamma(X)$ consists of the terms that are actually explored by the algorithm at state $X$. We will assume that $\Gamma$ contains $\mathsf{true}$, $\mathsf{false}$ and all subterms of its members; the subterms of a term $f(t_1, \ldots, t_j)$ need to be evaluated before location $f(t_1, \ldots, t_j)$ can be accessed. An algorithm satisfying Postulates I, II-lib and III-exact will be called *exacting*.

### 5.3 Explore Terms of ASMs

The explore sets of ASMs are defined in the most natural way. If $U$ is a set of terms, let $\bar{U}$ be the closure of $U \cup \{\mathsf{true}, \mathsf{false}\}$ under subterms. We have:

- $\Gamma_{f(s_1,\ldots,s_n):=t}(X) = \overline{\{s_1, \ldots, s_n, t\}}$;
- $\Gamma_{[P_1 \parallel \cdots \parallel P_n]}(X) = \Gamma_{P_1}(X) \cup \cdots \cup \Gamma_{P_n}(X)$; and
- $\Gamma_{\textbf{if}\, C\, \textbf{then}\, P}(X) = \overline{\{C\}} \cup \Gamma_P(X)$ if $X \models C$ and just $\overline{\{C\}}$, otherwise.

Thus $\Gamma_P(X)$ contains all conditions that ASM $P$ tests at state $X$, and all terms that occur in proposed updates.

**Theorem 2.** *Every ASM with explore sets as indicated is an exacting algorithm.*

*Proof.* Induction on ASM programs. □

## 6 Exacting Algorithms

**Theorem 3.**

1. *Every exacting algorithm with no partial basic functions in its states is also classical.*
2. *Every classical algorithm can be equipped with explore sets so as to be exacting.*

*Proof.*
1. The claim is obvious.
2. Given a classical algorithm, define $\Gamma(X)$ to be the set of all critical terms (regardless of the state $X$). □

A classical algorithm can often be equipped with explore sets in more than one way so as to be exacting. There is always a trivial way used in the proof above. But usually the algorithm will explore fewer terms than that in some (or even all) states, so a smaller $\Gamma(X)$ can be used.

A set $V$ of states is *uniform* if all states in $V$ have the same explore set, that is, if $\Gamma(X) = \Gamma(Y)$ for all $X, Y \in V$. For any set $V$ of states, let $\Gamma(V)$ denote the *shared* explore terms $\bigcap_{X \in V} \Gamma(X)$. We say that $V$ is *agreeable* if all states in $V$ agree on the values of all their shared explore terms, that is, if $X =_{\Gamma(V)} Y$ for all $X, Y \in V$. It stands to reason that agreeable states engender uniform behavior, because the algorithm has no way of distinguishing between them.

**Theorem 4.** *For any exacting algorithm, agreeability of a set of states implies its uniformity.*

*Proof.* By contradiction, suppose that, despite $V$'s agreeability, not all states in $V$ have the same explore set. Without loss of generality, let $t \in \Gamma(X)$ be a minimal explore term for some $X \in V$ that is not also an explore term for all other states in $V$ (minimal with respect to $\prec_X$), and let $Y \in V$ be a state such that $t \notin \Gamma(Y)$. By Discrimination, there is an $s \in \Gamma(X)$ such that $s \prec_X t$ and with different truth values in $X$ and $Y$. By agreeability, $s \notin \Gamma(V)$. But then $s$ must be a smaller choice of an explore term for $X$ than is $t$, since perforce $s \notin \Gamma(Z)$ for some $Z \in V$.

By Determination, we also have the following:

**Corollary 1.** *For any exacting algorithm, agreeability of a set $V$ of states implies that $\Delta(X) = \Delta(Y)$ for all $X, Y \in V$.*

In a sense, the Discrimination requirement is equivalent to the requirement that "agreeability implies uniformity". The latter requirement does not involve any ordering of explore terms.

**Theorem 5.** *Consider an alternative definition of exacting algorithms where the Discrimination requirement is replaced with the requirement that every agreeable set of states is uniform. The alternative definition is equivalent to the original definition.*

*Proof.* One direction is proved in the previous theorem. It remains to prove that an arbitrary alternative exacting algorithm satisfies Discrimination. Let $\mathcal{S}$ be the set of states of the algorithm.

For each $X \in \mathcal{S}$, we define a partial order $\prec_X$ on $\Gamma(X)$. Explore terms that are shared by all states are smallest, because they are always needed. Next come those terms that are shared by all states that agree with $X$ on the values of the lowest tier, $\Gamma(\mathcal{S})$, of terms. And so on. Thus, the ordering $\prec_X$, as a set of ordered pairs, is $L_X(\mathcal{S})$, where $L_X(V)$ is an ordering that discriminates $X$ from other states in $V$. When $V$ is uniform, $L_X(V) := \varnothing$; otherwise,

$$L_X(V) := (\Gamma(V) \times (\Gamma(X) \setminus \Gamma(V))) \cup L_X(\{Y \in V \mid Y =_{\Gamma(V)} X\}).$$

This recursion is bound to terminate, because $\Gamma(X) \setminus \Gamma(V)$ gets continually smaller. To see why, note that $X \in V$ always, so $\Gamma(V) \subseteq \Gamma(X)$. When $V$ is not uniform, it cannot be agreeable, so there is an $s \in \Gamma(V)$ over which states in $V$ disagree. But, by construction, all of $V$ agrees on all terms in the previous $\Gamma(V)$.

Now consider any $t \in \Gamma(X) \setminus \Gamma(Y)$ for a $Y \in \mathcal{S}$. Initially, $t \notin \Gamma(V) = \Gamma(\mathcal{S})$, whereas $t \in \Gamma(V) = \Gamma(X)$ at the end of the recursion, so $Y$ is not in the final argument $V$. At the point when $Y$ is removed from $V$, there must be an $s \in \Gamma(V)$ that discriminates between $X$ and $Y$. By construction, $s \prec_X t$. □

Equation 1 gives an apparently weaker form of "agreeability implies uniformity". It turns out that it is strictly weaker and insufficient to replace Discrimination.

## 7 Exacting ASM Representation Theorem

Two exacting algorithms $P$ and $Q$ are *behaviorally equivalent* if they operate over the same states, have the same initial states, the same state transitions, and explore the same terms at every state. By Corollary 1, for all states $X$, we have also $\Delta_P(X) = \Delta_Q(X)$. Unless the order in which locations are explored affects what is actually explored in a given state, we do not care about the precise order of exploration, nor about the number of times a location is accessed.

Note that, if exacting algorithms $P$ and $Q$ are behaviorally equivalent and $P$ hangs during the exploration of a state $X$ then so does $Q$, for the simple reason that they evaluate exactly the same terms. In the real world, a program may hang for various reasons, e.g. because the internet connection is poor. We abstract from such details in this theoretical study. An exacting algorithm hangs only because it attempts to evaluate an undefined term. Call a terminal state $X$ of an exacting algorithm $P$ *hanging* if $P$ hangs at $X$. It follows that behaviorally equivalent exacting algorithms have exactly the same hanging states.

**Theorem 6.** *Every exacting algorithm has a behaviorally equivalent clash-free ASM.*

In the beginning, we mentioned Kolmogorov's posit: "Algorithms compute in steps of bounded complexity". How do you measure the complexity of a step? A most natural non-numerical measure is the set of terms actually explored during the step. Combined with the cost of function calls, it leads to a natural numerical measure. In both cases, ASMs preserve the step complexity.

And the theorem shows that abstract state machines are adequate to emulate algorithms that may hang.

## 8 Discussion

We have shown that every exacting algorithm can be step-by-step emulated by an abstract state machine that at no state attempts to apply equality or another functions to more values than does the algorithm. This strengthens the thesis, propounded in [12], that abstract state machines faithfully model any and all sequential algorithms and bolsters the belief that the sequential postulates capture all sequential algorithms regardless of which model of computation they may be expressed in, including "continuous-space" algorithms.

"Continuous time" processes await further research. The easing of the requirements on fully-defined equality and other functions also lends strong support to the contention – put forth in [8,10] – that the Church-Turing Thesis is provably true from first principles. In addition to the sequential postulates, the arguments require that initial states contain only free constructors and functions that can be programmed from them (plus input). Our refinement of the ASM Representation Theorem strengthens those results by showing that the simulation of an algorithm, having no (unprogrammable) oracles, by an effective abstract state machine need not involve any operations not available to the original algorithm. It also follows from this work that there is no harm in incorporating partial operations in the initial states of effective algorithms, as long as they too can be computed effectively (whenever defined). Even with this relaxation of the limitations on initial states, it remains provable that no super-recursive function can be computed algorithmically.

## References

1. ASM Michigan Webpage `http://www.eecs.umich.edu/gasm/`, maintained by J. K. Huggins. Viewed June 4, 2010.
2. Bernot G., Bidoit M., Choppy C.: Abstract data types with exception handling. Theoret. Comp. Sci. 46, 13–45 (1986).
3. Bishop E.: Foundations of Constructive Analyis. McGrow-Hill (1967).
4. Blum L., Shub M., Smale S.: On a theory of computation and complexity over the real numbers. Bulletin of Amer. Math. Soc. (NS) 21, 1–46 (1989).
5. Blass A., Gurevich Y.: Algorithms vs. Machines. Bull. European Assoc. Theoret. Comp. Sci. 77, 96–118 (2002). Reprinted in: Paun G. et al. (eds.) Current Trends in Theoretical Computer Science: The Challenge of the New Century, vol. 2, 215–236 (2004).
6. Blass A., Gurevich Y.: Algorithms: A Quest for Absolute Definitions. Bull. Euro. Assoc. for Theor. Computer Sci. 81, 195–225 (2003). Reprinted in Current Trends in Theoretical Computer Science, 283–311, World Scientific (2004); and in Olszewski A. et al. (eds.) Church's Thesis After 70 Years, 24–57, Ontos (2006).
7. Börger E., Stärk R.: Abstract State Machines, Springer (2003).
8. Boker U., Dershowitz N.: The Church-Turing Thesis over arbitrary domains. In: Pillars of Computer Science, LNCS 4800, 199–229, Springer (2008).
9. xrc (exact reals in C). `http://keithbriggs.info/xrc.html`, viewed on June 4, 2010.

10. Dershowitz N., Gurevich Y.: A natural axiomatization of computability and proof of Church's Thesis. Bulletin of Symbolic Logic 14, 299–350 (2008).
11. Gowland P., Lester D.: A survey of exact arithmetic implementations. In: Blanck J. et al. (eds.) Computability and Complexity in Analysis, 4th International Workshop, CCA 2000, LNCS 2064, 30–47, Springer (2001).
12. Gurevich Y.: Evolving algebras 1993: Lipari guide. In: Börger E. (ed.), Specification and Validation Methods, 9–36, Oxford University Press (1995).
13. Gurevich Y.: Sequential abstract state machines capture sequential algorithms. ACM Transactions on Computational Logic 1, 77–111 (2000).
14. Kolmogorov A.N.: On the concept of algorithm. Uspekhi Matematicheskikh Nauk 8:4, 175–176 (1953) (in Russian). English version in [27, 18–19].
15. Korovina, M.: Gandy's theorem for abstract structures without the equality test. Proc. 10th Internat. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, LNCS 2850, 290–301, Springer (2003).
16. Kushner B.A.: Lectures on Constructive Mathematical Analysis. Translations of Mathematical Monographs, vol. 60, American Mathematical Society (1984). The Russian original published by Nauka in 1973.
17. Lambov, B.: The RealLib Project. `http://www.brics.dk/~barnie/RealLib/`, viewed on June 4, 2010.
18. Markov, A.A.: The Theory of Algorithms. American Mathematical Society Translations, series 2, 15, 1–14 (1960).
19. Meseguer J., Roşu G.: A total approach to partial algebraic specification. In: ICALP 2002, LNCS 2380, 572–584, Springer (2002).
20. Moschovakis Y.: On founding the theory of algorithms. In: Dales H.G., Olivieri G. (eds.) Truth in Mathematics, 71–104, Clarendon Press, Oxford (1998).
21. Moschovakis Y.: What is an algorithm? In: Engquist B., Schmid W. (eds.) Mathematics Unlimited — 2001 and Beyond, 919–936, Springer (2001).
22. Müller N.: iRRAM - Exact Arithmetic in C++. `http://www.informatik.uni-trier.de/iRRAM/`, viewed on June 4, 2010.
23. Naughton T.J.: Continuous-space model of computation is Turing universal. Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, Volume 4109, 121–128 (2000).
24. Nowack A.: Complexity theory via abstract state machines. Master's thesis, RWTH-Aachen (2000).
25. Spec Explorer. `http://msdn.microsoft.com/en-us/devlabs/ee692301.aspx`, viewed on June 04, 2010.
26. Computable Real Numbers, `http://www.haible.de/bruno/MichaelStoll/reals.html`, viewed on June 4, 2010.
27. Uspensky V.A., Semenov A.L.: Algorithms: Main Ideas and Applications. Kluwer (1993).
28. Weihrauch K.: Computable Analysis — An introduction. Springer (2000).
29. Winskel G.: Event Structures — Lecture Notes for the Advanced Course on Petri Nets. Univ. of Cambridge Computer Lab Tech. Report 95, UCAM-CL-TR-95 (1986).