The Generic Model of Computation

Nachum Dershowitz
School of Computer Science
Tel Aviv University
Tel Aviv, Israel
nachum.dershowitz@cs.tau.ac.il

Over the past two decades, Yuri Gurevich and his colleagues have formulated axiomatic foundations for the notion of *algorithm*, be it classical, interactive, or parallel, and formalized them in the new generic framework of *abstract state machines*. This approach has recently been extended to suggest a formalization of the notion of *effective* computation over arbitrary countable domains. The central notions are summarized herein.

1 Background

Abstract state machines (ASMs), invented by Yuri Gurevich [24], constitute a most general model of computation, one that can operate on any desired level of abstraction of data structures and native operations. All (ordinary) models of computation are instances of this one generic paradigm. Here, we give an overview of the foundational considerations underlying the model (cobbled together primarily from [18, 3, 12]).[1]

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

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

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

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

This sorting program is not partial to any particular representation of the natural numbers 1, 2, etc., which are being used to index F. Whether an implementation uses natural language, or decimal numbers,

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

To appear in EPTCS.

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

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

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

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

or binary strings is immaterial, as long as addition behaves as expected (and equality and disequality, too). Furthermore, the program will work regardless of the domain from which the values of F are drawn (be they integers, reals, strings, or what not), so long as means are provided for evaluating the inequality ($>$) relation.

Another simple ASM program is shown in Algorithm 2. This is a standard bisection search for the root of a function, as described in [22, Algorithm #4]. The point is that this abstract formulation is, as the author of [22] wrote, “applicable to any continuous function” over the reals—including ones that cannot be programmed.

What is remarkable about ASMs is that this very simple model of computation suffices to precisely capture the behavior of the whole class of ordinary algorithms over any domain. The reason is that, by virtue of the abstract state machine (ASM) representation theorem of [25] (Theorem 2 below), any algorithm that satisfies three very natural “Sequential Postulates” can be *step-by-step, state-for-state* emulated by an ASM. Those postulates, articulated in Section 2, formalize the following intuitions: (I) an algorithm is a state-transition system; (II) given the algorithm, state information determines future transitions and can be captured by a logical structure; and (III) state transitions are governed by the values of a finite and input-independent set of terms.

The significance of the Sequential Postulates lies in their comprehensiveness. They formalize which features exactly characterize a classical algorithm in its most abstract and generic manifestation. Programs of all models of effective, sequential computation satisfy the postulates, as do idealized algorithms for computing with real numbers (e.g. Algorithm 2), or for geometric constructions with compass and straightedge (see [34] for examples of the latter).

Abstract state machines are a computational model that is not wedded to any particular data representation, in the way, say, that Turing machines manipulate strings using a small set of tape operations. The Representation Theorem, restated in Section 3, establishes that ASMs can express and precisely emulate any and all algorithms satisfying the premises captured by the postulates. For any such algorithm, there is an ASM program that describes precisely the same state-transition function, state after state, as does the algorithm. In this sense, ASMs subsume all other computational models.

It may be informative to note the similarity between the form of an ASM, namely, a single repeated loop of a set of generalized assignments nested within conditionals with the “folk theorem” to the effect that any flowchart program can be converted to a single loop composed of conditionals, sequencing, and assignments, with the aid of some auxiliary variables (see [29]). Parallel composition gives ASMs the ability to perform multiple actions sans extra variables, and to capture all that transpires in a single step of any algorithm.

This versatility of ASMs is what makes them so ideal for both specification and prototyping. Indeed, ASMs have been used to model all manner of programming applications, systems, and languages, each on the precise intended level of abstraction. See [13] and the ASM website (`http://www.eecs.umich.edu/gasm`) for numerous exemplars. ASMs provide a complete means of describing algorithms, whether or not they can be implemented effectively. On account of their abstractness, one can express generic algorithms, like our bisection search for arbitrary continuous real-valued functions, or like Gaussian elimination, even when the field over which it is applied is left unspecified. AsmL [26], an executable specification language based on the ASM framework, has been used in industry, in particular for the behavioral specification of interfaces (see, for example, [1]).

Church's Thesis asserts that the recursive functions are the only numeric functions that can be effectively computed. Similarly, Turing's Thesis stakes the claim that any function on strings that can be mechanically computed can be computed, in particular, by a Turing machine. More generally, one additional natural hypothesis regarding the describability of initial states of algorithms, as explained in Section 5, characterizes the effectiveness of any model of computation, operating over any (countable) data domain (Theorem 4).

On account of the ability of ASMs to precisely capture single steps of any algorithm, one can infer absolute bounds on the complexity of algorithms under arbitrary effective models of computation, as will be seen (Theorem 6) at the end of Section 5.

2 Sequential Algorithms

The Sequential Postulates of [25] regarding algorithmic behavior are based on the following key observations:

- A state should contain *all* the relevant information, apart from the algorithm itself, needed to determine the next steps. For example, the "instantaneous description" of a Turing machine computation is just what is needed to pick up a machine's computation from where it has been left off; see [38]. Similarly, the "continuation" of a Lisp program contains all the state information needed to resume its computation. First-order structures suffice to model all salient features of states. Compare [32, pp. 420–429].
- The values of programming variables, in and of themselves, are meaningless to an algorithm, which is implementation independent. Rather, it is relationships between values that matter to the algorithm. It follows that an algorithm should work equally well in isomorphic worlds. Compare [19, p. 128]. An algorithm can—indeed, can only—determine relations between values stored in a state via terms in its vocabulary and equalities (and disequalities) between their values.
- Algorithms are expressed by means of finite texts, making reference to only finitely many terms and relations among them. See, for example, [31, p. 493].

The three postulates given below (from [25], modified slightly as in [4, 5, 6, 3]) assert that a classical algorithm is a state-transition system operating over first-order structures in a way that is invariant under isomorphisms. An algorithm is a prescription for updating states, that is, for changing some of the interpretations given to symbols by states. The essential idea is that there is a fixed finite set of terms that refer (possibly indirectly) to locations within a state and which suffice to determine how the state changes during any transition.

2.1 Sequential Time

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

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

- *A nonempty set*[2] $\mathscr{S}$ *of* states *and a nonempty subset* $\mathscr{S}_0 \subseteq \mathscr{S}$ *of* initial *states.*
- *A partial* next-state *transition function* $\tau : \mathscr{S} \rightharpoonup \mathscr{S}$.

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

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

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

This postulate is meant to exclude formalisms, such as [21, 33], in which the result of a computation—or the continuation of a computation—may depend on (the limit of) an infinite sequence of preceding (finite or infinitesimal) steps. Likewise, processes in which states evolve continuously (as in analog processes, like the position of a bouncing ball), rather than discretely, are eschewed.

Though it may appear at first glance that a recursive function does not fit under the rubric of a state-transition system, in fact the definition of a traditional recursive function comes together with a computation rule for evaluating it. As Rogers [36, p. 7] writes, “We obtain the computation uniquely by working from the inside out and from left to right”.

2.2 Abstract State

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

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

- *If* X *is a state of the algorithm, then any structure* Y *that is isomorphic to* X *is also a state, and* Y *is initial or terminal if* X *is initial or terminal, respectively.*
- *Transitions preserve the domain; that is,* $\text{Dom}\ \tau(X) = \text{Dom}\ X$ *for every non-terminal state* X.

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

- *Transitions respect isomorphisms, so, if $\zeta : X \cong Y$ is an isomorphism of non-terminal states X,Y, then also $\zeta : \tau(X) \cong \tau(Y)$.*

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

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

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

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

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

It is possible to liberalize this postulate somewhat to allow the domain to grow or shrink, or for the vocabulary to be infinite or extensible, but such "enhancements" do not materially change the notion of algorithm. An extension to structures with partial operations is given in [3]; see Section 4.

2.3 Effective Transitions

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

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

Postulate III (Effective Transitions)[3] *For every algorithm, there is a finite set T of (ground)* critical terms *over the state vocabulary, such that states that agree on the values of the terms in T also share the same update sets. That is, $\Delta(X) = \Delta(Y)$, for any two states X,Y such that $[\![t]\!]_X = [\![t]\!]_Y$ for all $t \in T$. In particular, if one of X and Y is terminal, so is the other.*

[3]Or **Bounded Exploration**.

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

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

A careful analysis of the notion of algorithm in [25] and an examination of the intent of the founders of the field of computability in [18] demonstrate that the Sequential Postulates are in fact true of all ordinary, sequential algorithms, the (only) kind envisioned by the pioneers of the field. In other words, all *classical* algorithms satisfy Postulates I, II, and III. In this sense, the traditional notion of algorithm is precisely captured by these axioms.

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

2.4 Equivalent Algorithms

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

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

For a wide-ranging discussion of algorithm equivalence, see [2].

3 Abstract State Machines

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

3.1 Programs

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

For convenience, we show only a simple form of ASMs. Bear in mind, however, that much richer languages for ASMs are given in [24] and are used in practice [27].

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

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

Table 1: Update sets for sorting program.

of F. This program always terminates successfully, with $j = n = i+1$ and with the first n elements of F in nondecreasing order.

There are no hidden variables in ASMs. If some steps of an algorithm are intended to be executed in sequence, say, then the ASM will need to keep explicit track of where in the sequence it is up to.

3.2 Semantics

Unlike algorithms, which are observed to either change the value of a location in the current state, or not, an ASM might "update" a location in a *trivial* way, giving it the same value it already has. Also, an ASM might designate two conflicting updates for the same location, what is called a *clash*, in which case the standard ASM semantics are to cause the run to fail (just as real-world programs might abort). An alternative semantics is to imagine a nondeterministic choice between the competing values. (Both were considered in [24].) Here, we prefer to ignore both nondeterminism and implicit failure, and tacitly presume that an ASM never involves clashes, albeit this is an undecidable property.

To take the various possibilities into account, a *proposed* update set $\Delta^+_P(X)$ (cf. [4]) for an ASM P may be defined in the following manner:

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

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

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

For terminal states X, the update set $\Delta(X)$ is $\perp$, to signify that there is no next state. For non-terminal X, $\Delta(X)$ is the set of non-trivial updates in $\Delta^+(X)$. The update sets for the sorting program (Algorithm 1) are shown in Table 1, with the subscript in $[\![\cdot]\!]_X$ omitted. For example, if state X is such that $n = 2$, $i = 0$,

$j = 1$, $F(0) = 1$, and $F(1) = 0$, then (per row 2) $\Delta^+(X) = \{F(0) \mapsto 0, F(1) \mapsto 1, j \mapsto 2\}$. For this X, $\Delta(X) = \Delta^+(X)$, and the next state $X' = \tau(X)$ has $i = 0$ (as before), $j = 2$, $F(0) = 0$ and $F(1) = 1$. After one more step (per row 1), in which F is unchanged, the algorithm reaches a terminal state, $X'' = \tau(X')$, with $j = n = i + 1 = 2$. Then (by row 0), $\Delta^+(X'') = \varnothing$ and $\Delta(X'') = \bot$.

4 The Representation Theorem

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

The converse is of greater significance:

Theorem 2 (Representation [25, Theorem 6.13]) *Every classical algorithm, in the sense of Definition 1, has a behaviorally equivalent ASM, with the exact same states and state-transition function.*

The proof of this representation theorem constructs an ASM that contains conditions involving equalities and disequalities between critical terms. Closure under isomorphisms is an essential ingredient for making it possible to express any algorithm in the language of terms.

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

In [3], the analysis and representation theorem have been refined for algorithms employing truly partial operations, operations that cause an algorithm to hang when an operation is attempted outside its domain of definition (rather than return `undef`). The point is that there is a behaviorally equivalent ASM that never attempts to access locations in the state that are not also accessed by the given algorithm. Such partial operations are required in the next section.

5 Effective Algorithms

The Church-Turing Thesis [30, Thesis I†] asserts that standard models capture effective computation. Specifically:

> All effectively computable numeric (partial) functions are (partial) recursive.
> All (partial) string functions can be computed by a Turing machine.

We say that an algorithm *computes* a partial function $f : D^k \rightharpoonup D$ if there are *input* states $\mathscr{I} \subseteq \mathscr{S}_0$, with particular locations for input values, such that running the algorithm results in the correct output values of f. Specifically:

- The domain of each input state is D. There are k terms such that their values in input states cover all tuples in D^k. Other than that, input states all agree on the values of all other terms.

- For all input values $\overline{a}$, the corresponding input state leads, via a sequence of transitions τ, to a terminal state in which the value of a designated term t (in the vocabulary of the algorithm) is $f(\overline{a})$ whenever the latter is defined, and leads to an infinite computation whenever it is not.

To capture what it is that makes a sequential algorithm mechanically computable, we need for input states to be finitely representable. Accordingly, we insist that they harbor no information beyond the means to reach domain values, plus anything that can be derived therefrom.

We say that function symbols $\mathscr{C}$ *construct* domain D in state X if X assigns each value in D to exactly one term over $\mathscr{C}$, so restricting X to $\mathscr{C}$ gives a free Herbrand algebra. For example, the domain of the sorting algorithm, consisting of integers and Booleans, can be constructed from $0, \mathsf{true}, \mathsf{false}, \mathsf{undef}$, and a "successor" function (call it c) that takes non-negative integers (n) to the predecessor of their negation ($-n-1$) and negative integers ($-n$) to their absolute value (n).

Postulate III ensures that the transition function is describable by a finite text, and—in particular–by the text of ASM. For an algorithm to be effective, its states must also be finitely describable.

Definition 3 (Effectiveness)

1. *A state is* effective *if it includes constructors for its domain, plus operations that are almost everywhere the same, meaning that all but finitely-many locations (these can hold input values) have the same default value (such as* undef*).*
2. *A classical algorithm is* effective *if its initial states are.*
3. *Moreover, effective algorithms can be bootstrapped: A state is effective also if its vocabulary can be enriched to* $\mathscr{C} \uplus \mathscr{G}$ *so that* $\mathscr{C}$ *constructs its domain, while every (total or partial) operation in* $\mathscr{G}$ *is computed by an effective algorithm over those constructors.*
4. *A* model (of computation)*, that is, a set of algorithms with shared domain(s), is* effective *if all its algorithms are, via the* same *constructors.*

This effectiveness postulate excludes algorithms with ineffective oracles, such as the halting function. Having only free constructors at the foundation precludes the hiding of potentially uncomputable information by means of equalities between distinct representations of the same domain element.

This is the approach to effectiveness advocated in [11], extended to include partial functions in states, as in [3]. For each $n \geq 1$, our sorting algorithm is effective in this sense, since addition ($+$) of the natural numbers and comparisons ($>$) of integers, operations that reside in its initial states, can be programmed from the above-mentioned constructors ($0, \mathsf{true}, \mathsf{false}, \mathsf{undef}, c$).

In particular, partial-recursion for natural numbers and Turing machines for strings form effective models [11]. Furthermore, it is shown in [12] that three prima facie different definitions of effectiveness over arbitrary domains, as proposed in [11, 18, 35], respectively, comprise exactly the same functions, strengthening the conviction that the essence of the underlying notion of computability has in fact been captured.

Theorem 4 (Church-Turing Thesis [11]) *For every effective model, there is a representation of its domain values as strings, such that its algorithms are each simulated by some Turing machine.*

Call an effective computational model *maximal* if adding any function to those that it computes results in a set of functions that cannot be simulated by any effective model. Remarkably (or perhaps not), there is exactly one such model:

Theorem 5 (Effectiveness [12, Theorem 4]) *The set of partial recursive functions (and likewise the set of Turing-computable string functions) is the unique maximal effective model, up to isomorphism, over any countable domain.*

We have recently extended the proof of the Church-Turing Thesis and demonstrated the validity of the widely believed *Extended Church-Turing Thesis*:

Theorem 6 (Extended Church-Turing Thesis [17]) *Every effective algorithm can be polynomially simulated by a Turing machine.*

6 Conclusion

We have dealt herein with the classical type of algorithms, that is to say, with the “small-step” (meaning, only bounded parallelism) “sequential-time” (deterministic, no intra-step interaction with the outside world) case. Abstract state machines can faithfully emulate any algorithm in this class, as we have seen in Theorem 2. Furthermore, we have characterized the distinction between effective algorithms and their more abstract siblings in Theorem 4.

There are various “declarative” styles of programming for which the state-transition relation is implicit, rather than explicit as it is for our notion of algorithm. For such programs to be algorithms in the sense of Definition 1, they would have to be equipped with a specific execution mechanism, like the one for recursion mentioned above. For Prolog, for example, the mechanism of unification and the mode of search would need to be specified [14].

The abstract-state-machine paradigm can be extended to handle more modern notions:

- When desired, an algorithm can make an explicit distinction between successful and failing terminal states by storing particular values in specific locations of the final state. Alternatively, one may declare failure when there is a conflict between two or more enabled assignments. See [24].
- There is no difficulty in allowing for nondeterminism, that is, for a multivalued transition function. If the semantics are such that a choice is made between clashing assignment statements, then transitions are indeed nondeterministic. See [24, 28].
- More general forms of nondeterminism can be obtained by adding a choice command of some sort to the language. See [24].
- Nothing needs to be added to the syntax of ASMs to apply to cases for the environment provides input incrementally. One need only imagine that the environment is allowed to modify the values of some (specified) set of locations in the state between machine steps. See [24].
- In [4, 5, 6], the analysis of algorithms was extended to the case when an algorithm interacts with the outside environment during a step, and execution waits until all queries of the environment have been responded to.
- In [8, 9], all forms of interaction are handled.
- In [7], the analysis was extended to massively parallel algorithms.
- Distributed algorithms are handled in [24, 20].
- The fact that ASMs can emulate algorithms step-for-step facilitates reasoning about the complexity of algorithms, as for Theorem 6 above. Parallel ASMs have been used for studying the complexity of algorithms over unordered structures. See [10, 37].
- Quantum algorithms have been modeled by ASMs in [23].
- Current research includes an extension of the framework for hybrid systems, combining discrete (sequential steps) and analog (evolving over time) behaviors [15, 16].

Acknowledgements

I thank Yuri Gurevich and Nikolaj Bjørner for their perspicacious suggestions, the referees for their questions, and Evgenia Falkovich for her help.

References

[1] Mike Barnett & Wolfram Schulte (2001): *The ABCs of Specification: AsmL, Behavior, and Components*. *Informatica (Slovenia)* 25(4), pp. 517–526. Available at `http://research.microsoft.com/pubs/73061/TheABCsOfSpecification(Informatica2001).pdf` (viewed June 7, 2009).

[2] Andreas Blass, Nachum Dershowitz & Yuri Gurevich (2009): *When are Two Algorithms the Same? Bulletin of Symbolic Logic* 15(2), pp. 145–168, doi:`10.2178/bsl/1243948484`. Available at `http://nachum.org/papers/WhenAreTwo.pdf` (viewed Mar. 27, 2011).

[3] Andreas Blass, Nachum Dershowitz & Yuri Gurevich (2010): *Exact Exploration and Hanging Algorithms*. In: *Proceedings of the 19th EACSL Annual Conferences on Computer Science Logic (Brno, Czech Republic)*, Lecture Notes in Computer Science, Springer, Berlin, Germany, pp. 140–154, doi:`10.1007/978-3-642-15205-4_14`. Available at `http://nachum.org/papers/HangingAlgorithms.pdf` (viewed May 27, 2011); longer version at `http://nachum.org/papers/ExactExploration.pdf` (viewed May 27, 2011).

[4] Andreas Blass & Yuri Gurevich (2006): *Ordinary Interactive Small-Step Algorithms, Part I*. *ACM Transactions on Computational Logic* 7(2), pp. 363–419, doi:`10.1145/1131313.1131320`. Available at `http://tocl.acm.org/accepted/blass04.ps` (viewed May 21, 2009).

[5] Andreas Blass & Yuri Gurevich (2007): *Ordinary Interactive Small-Step Algorithms, Part II*. *ACM Transactions on Computational Logic* 8(3), doi:`10.1145/1243996.1243998`. Article 15. Available at `http://tocl.acm.org/accepted/blass2.pdf` (viewed May 21, 2009).

[6] Andreas Blass & Yuri Gurevich (2007): *Ordinary Interactive Small-Step Algorithms, Part III*. *ACM Transactions on Computational Logic* 8(3), doi:`10.1145/1243996.1243999`. Article 16. Available at `http://tocl.acm.org/accepted/250blass.pdf` (viewed May 21, 2009).

[7] Andreas Blass & Yuri Gurevich (2008): *Abstract State Machines Capture Parallel Algorithms: Correction and Extension*. *ACM Transactions on Computation Logic* 9(3), doi:`10.1145/1352582.1352587`. Article 19. Available at `http://research.microsoft.com/en-us/um/people/gurevich/Opera/157-2.pdf` (viewed Aug. 11, 2010).

[8] Andreas Blass, Yuri Gurevich, Dean Rosenzweig & Benjamin Rossman (2007): *Interactive Small-Step Algorithms, Part I: Axiomatization*. *Logical Methods in Computer Science* 3(4), doi:`10.2168/LMCS-3(4:3)2007`. Paper 3. Available at `http://research.microsoft.com/~gurevich/Opera/176.pdf` (viewed June 5, 2009).

[9] Andreas Blass, Yuri Gurevich, Dean Rosenzweig & Benjamin Rossman (2007): *Interactive Small-Step Algorithms, Part II: Abstract State Machines and the Characterization Theorem*. *Logical Methods in Computer Science* 4(4), doi:`10.2168/LMCS-3(4:4)2007`. Paper 4. Available at `http://arxiv.org/pdf/0707.3789v2` (viewed July 17, 2011).

[10] Andreas Blass, Yuri Gurevich & Saharon Shelah (2002): *On Polynomial Time Computation over Unordered Structures*. *Journal of Symbolic Logic* 67(3), pp. 1093–1125, doi:`10.2178/jsl/1190150152`. Available at `http://research.microsoft.com/en-us/um/people/gurevich/Opera/150.pdf` (viewed July 13, 2011).

[11] Udi Boker & Nachum Dershowitz (2008): *The Church-Turing Thesis over Arbitrary Domains*. In Arnon Avron, Nachum Dershowitz & Alexander Rabinovich, editors: *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, *Lecture Notes in Computer Science*

4800, Springer, pp. 199–229, doi:10.1007/978-3-540-78127-1_12. Available at http://nachum.org/papers/ArbitraryDomains.pdf (viewed Aug. 11, 2010).

[12] Udi Boker & Nachum Dershowitz (2010): *Three Paths to Effectiveness*. In Andreas Blass, Nachum Dershowitz & Wolfgang Reisig, editors: *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, *Lecture Notes in Computer Science* 6300, Springer, Berlin, Germany, pp. 36–47, doi:10.1007/978-3-642-15025-8_7. Available at http://nachum.org/papers/ThreePathsToEffectiveness.pdf (viewed Aug. 11, 2010).

[13] Egon Börger (2002): *The Origins and the Development of the ASM Method for High Level System Design and Analysis*. *Journal of Universal Computer Science* 8(1), pp. 2–74, doi:10.3217/jucs-008-01-0002. Available at http://www.jucs.org/jucs_8_1/the_origins_and_the/Boerger_E.pdf (viewed June 17, 2009).

[14] Egon Börger & Dean Rosenzweig (1995): *A Mathematical Definition of Full Prolog*. *Science of Computer Programming* 24, pp. 249–286, doi:10.1016/0167-6423(95)00006-E. Available at ftp://www.eecs.umich.edu/groups/gasm/prolog.pdf (viewed July 17, 2011).

[15] Olivier Bournez & Nachum Dershowitz (2010): *Foundations of Analog Algorithms*. In: *Proceedings of the Third International Workshop on Physics and Computation (P&C)*, Nile River, Egypt, pp. 85–94. Available at http://nachum.org/papers/Analog.pdf (viewed May 27, 2011).

[16] Olivier Bournez, Nachum Dershowitz & Evgenia Falkovich (2012): *Towards an Axiomatization of Simple Analog Algorithms*. In Manindra Agrawal, S. Barry Cooper & Angsheng Li, editors: *Proceedings of the 9th Annual Conference on Theory and Applications of Models of Computation (TAMC 2012, Beijing, China)*, *Lecture Notes in Computer Science* 7287, Springer Verlag, pp. 525–536. Available at http://dx.doi.org/10.1007/978-3-642-29952-0_49. Available at http://nachum.org/papers/SimpleAnalog.pdf (viewed July 11, 2012).

[17] Nachum Dershowitz & Evgenia Falkovich (2011): *A Formalization and Proof of the Extended Church-Turing Thesis*. In: *Proceedings of the Seventh International Workshop on Developments in Computational Models (DCM 2011, July 2012, Zurich, Switzerland)*, Electronic Proceedings in Theoretical Computer Science. Available at http://nachum.org/papers/ECCT.pdf (viewed July 15, 2011).

[18] Nachum Dershowitz & Yuri Gurevich (2008): *A Natural Axiomatization of Computability and Proof of Church's Thesis*. *Bulletin of Symbolic Logic* 14(3), pp. 299–350, doi:10.2178/bsl/1231081370. Available at http://nachum.org/papers/Church.pdf (viewed Apr. 15, 2009).

[19] Robin Gandy (1980): *Church's Thesis and Principles for Mechanisms*. In: *The Kleene Symposium*, *Studies in Logic and the Foundations of Mathematics* 101, North-Holland, pp. 123–148, doi:10.1016/S0049-237X(08)71257-6.

[20] Andreas Glausch & Wolfgang Reisig (2009): *An ASM-Characterization of a Class of Distributed Algorithms*. In Jean-Raymond Abrial & Uwe Glässer, editors: *Rigorous Methods for Software Construction and Analysis*, *Lecture Notes in Computer Science* 5115, Springer, Berlin, pp. 50–64, doi:10.1007/978-3-642-11447-2_4. Available at http://www2.informatik.hu-berlin.de/top/download/publications/GlauschR2007_dagstuhl.pdf (viewed Aug. 11, 2010).

[21] E. Mark Gold (1965): *Limiting Recursion*. *J. Symbolic Logic* 30(1), pp. 28–48, doi:10.2307/2270580.

[22] Saul Gorn (1960): *Algorithms: Bisection Routine*. *Communications of the ACM* 3(3), p. 174, doi:10.1145/367149.367173.

[23] Erich Grädel & Antje Nowack (2003): *Quantum Computing and Abstract State Machines*. In: *Proceedings of the 10th International Conference on Abstract State Machines: Advances in Theory and Practice (ASM '03; Taormina, Italy)*, Springer-Verlag, Berlin, pp. 309–323, doi:10.1007/3-540-36498-6_18. Available at http://www.logic.rwth-aachen.de/pub/graedel/GrNo-asm03.ps (viewed July 13, 2011).

[24] Yuri Gurevich (1995): *Evolving Algebras 1993: Lipari Guide*. In Egon Börger, editor: *Specification and Validation Methods*, Oxford University Press, pp. 9–36. Available at http://research.microsoft.com/~gurevich/opera/103.pdf (viewed Apr. 15, 2009).

[25] Yuri Gurevich (2000): *Sequential Abstract State Machines Capture Sequential Algorithms*. *ACM Transactions on Computational Logic* 1(1), pp. 77–111, doi:`10.1145/343369.343384`. Available at `http://research.microsoft.com/~gurevich/opera/141.pdf` (viewed Apr. 15, 2009).

[26] Yuri Gurevich, Benjamin Rossman & Wolfram Schulte (2005): *Semantic Essence of AsmL*. *Theoretical Computer Science* 343(3), pp. 370–412, doi:`10.1016/j.tcs.2005.06.017`. Available at `http://research.microsoft.com/~gurevich/opera/169.pdf` (viewed June 7, 2009).

[27] Yuri Gurevich, Wolfram Schulte & Margus Veanes (2001): *Toward Industrial Strength Abstract State Machines*. Technical Report MSR-TR-2001-98, Microsoft Research. Available at `http://research.microsoft.com/en-us/um/people/gurevich/opera/155.pdf` (viewed Aug. 11, 2010).

[28] Yuri Gurevich & Tatiana Yavorskaya (2006): *On Bounded Exploration and Bounded Nondeterminism*. Technical Report MSR-TR-2006-07, Microsoft Research. Available at `http://research.microsoft.com/~gurevich/opera/177.pdf` (viewed Apr. 15, 2009).

[29] David Harel (1980): *On Folk Theorems*. *Communications of the ACM* 23(7), pp. 379–389, doi:`10.1145/358886.358892`.

[30] Stephen C. Kleene (1967): *Mathematical Logic*. Wiley, New York.

[31] Stephen C. Kleene (1987): *Reflections on Church's Thesis*. *Notre Dame Journal of Formal Logic* 28(4), pp. 490–498, doi:`10.1305/ndjfl/1093637645`.

[32] Emil L. Post (1994): *Absolutely Unsolvable Problems and Relatively Undecidable Propositions: Account of an Anticipation*. In M. Davis, editor: *Solvability, Provability, Definability: The Collected Works of Emil L. Post*, Birkhaüser, Boston, MA, pp. 375–441. Unpublished paper, 1941.

[33] Hilary Putnam (1965): *Trial and Error Predicates and the Solution to a Problem of Mostowski*. *J. Symbolic Logic* 30(1), pp. 49–57, doi:`10.2307/2270581`.

[34] Wolfgang Reisig (2003): *On Gurevich's Theorem on Sequential Algorithms*. *Acta Informatica* 39(4), pp. 273–305, doi:`10.1007/s00236-002-0106-3`. Available at `http://www2.informatik.hu-berlin.de/top/download/publications/Reisig2003_ai395.pdf` (viewed Aug. 11, 2010).

[35] Wolfgang Reisig (2008): *The Computable Kernel of Abstract State Machines*. *Theoretical Computer Science* 409(1), pp. 126–136, doi:`10.1016/j.tcs.2008.08.041`. Draft available at `http://www2.informatik.hu-berlin.de/top/download/publications/Reisig2004_hub_tr177.pdf` (viewed Aug. 11, 2010).

[36] Hartley Rogers, Jr. (1966): *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York.

[37] Marc Spielmann (2000): *Abstract State Machines: Verification Problems and Complexity*. Ph.D. thesis, RWTH Aachen, Aachen, Germany. Available at `http://www-mgi.informatik.rwth-aachen.de/~spielmann/diss.pdf` (viewed July 13, 2011).

[38] Alan M. Turing (1936–37): *On Computable Numbers, With an Application to the Entscheidungsproblem*. *Proceedings of the London Mathematical Society* 42, pp. 230–265, doi:`10.1112/plms/s2-42.1.230`. Corrections in vol. 43 (1937), pp. 544-546. Reprinted in M. Davis (ed.), *The Undecidable*, Raven Press, Hewlett, NY, 1965. Available at `http://www.abelard.org/turpap2/tp2-ie.asp`.