Three Paths to Effectiveness

—Extended Abstract—

Udi Boker
School of Engineering and Computer Science, Hebrew University
Jerusalem 91904, Israel
udiboker@cs.huji.ac.il

Nachum Dershowitz*
School of Computer Science, Tel Aviv University
Ramat Aviv 69978, Israel
nachum.dershowitz@cs.tau.ac.il

July 27, 2009

Abstract

We compare three seemingly disparate notions of effectiveness of computational models operating over non-standard domains within the Abstract State Machine framework of Gurevich. We show that, though taking different routes, they all lead to the same concept.

1 Introduction

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. For models of computation that operate over arbitrary data structures, however, these two standard notions of what constitutes effectiveness may not be directly applicable.

Sequential algorithms—that is, deterministic algorithms without unbounded parallelism or (intra-step) interaction with the outside world—have been analyzed and formalized by Gurevich in [Gur00]. There it was proved that any algorithm satisfying three natural formal postulates (given below) can be emulated, step by step, by a program in a very general model of computation, called abstract state machines. But an algorithm, or abstract state machine program, need not yield an effective function. Gaussian elimination, for example, is a

*Supported in part by the Israel Science Foundation (grant no. 250/05).

perfectly well-defined algorithm over the real numbers, even though the reals cannot all be effectively represented and manipulated.

We adopt the necessary point of view that effectiveness is a notion applicable to collections of functions, rather than to single functions (cf. [Myh52]). A single function over an arbitrary domain cannot be classified as effective or ineffective [Mon60, Sha82], since its effectiveness depends on the context. A detailed discussion of this issue can be found in [BD08].

To capture what it is that makes a sequential algorithm mechanically computable, three different generic formalizations of effectiveness have recently been suggested:

- In [BD08], the authors base their notion of effectivity on finite constructibility. Initial data are inductively defined to be effective if it only contains a Herbrand universe in addition to some finite data, and to any function that can be shown constructible in the same way.
- In [DG08], Dershowitz and Gurevich require an injective mapping between the arbitrary domain and the natural numbers. Initial data are effective if they are tracked—under that representation—by recursive functions.
- In [Rei08], Reisig bases effectiveness on the natural congruence relation between vocabulary terms. Initial data are effective if the induced congruence between terms is Turing-computable.

Properly extending these approaches to a set of algorithms, it turns out that these three notions are essentially one and the same.

2 Algorithms

We work in the abstract state machine framework of [Gur00]. We begin by recalling Gurevich's Sequential Postulates, formalizing the following intuitions: (I) we are dealing with discrete deterministic state-transition systems; (II) the information in states suffices to determine future transitions and may be captured by logical structures that respect isomorphisms; and (III) transitions are governed by the values of a finite and input-independent set of terms. See [DG08] for historical support for the postulates.

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

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

A *terminal state* is one for which no transition is defined. Let $\mathcal{O} \subseteq \mathcal{S}$ denote the (possibly empty) set of terminal states. We write $x \rightsquigarrow_\tau x'$ when $x' = \tau(x)$. A *computation* is a finite or infinite chain $x_0 \rightsquigarrow_\tau x_1 \rightsquigarrow_\tau \cdots$ of states.

It may appear that a recursive function is not a state-transition system, but in fact the definition of a recursive function comes together with a computation rule for evaluating it. As Rogers [Rog66, p. 7] writes, "We obtain the computation uniquely by working from the inside out and from left to right".

Since transitions are functions, the states of an algorithm must contain all the information necessary to determine the future of a computation. They must contain a full "instantaneous description" of all relevant aspects of the current status of a computation.

Logical structures are ideal for capturing all the salient information stored in a state. All structures in this paper are over first-order finite vocabularies, have countably many elements in their domains (base sets), and interpret symbols as total operations. All relations are viewed as truth-valued functions, so we refer to structures as algebras. We always assume that structures include Boolean truth values, standard Boolean operations, and equality, and that vocabularies include symbols for these.

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

1. *If $x \in \mathcal{S}$ is a state of the algorithm, then any algebra y isomorphic to x is also a state in $\mathcal{S}$, and y is initial or terminal if x is initial or terminal, respectively.*

2. *Transitions τ preserve the domain; that is,* Dom $\tau(x) =$ Dom x *for every non-terminal state $x \in \mathcal{S} \setminus \mathcal{O}$.*

3. *Transitions respect isomorphisms, so, if $\zeta : x \cong y$ is an isomorphism of non-terminal states $x, y \in \mathcal{S} \setminus \mathcal{O}$, then $\zeta : \tau(x) \cong \tau(y)$.*

We refer to such states as "abstract", because the isomorphism requirement means that transitions do not depend in any essential way on the specific representation of the domain embodied in a given state.

Since a state x is an algebra, it interprets function symbols in $\mathcal{F}$, assigning a value $c \in$ Dom x to the "location" $f(a_1, \ldots, a_k)$ in x for every k-ary symbol $f \in \mathcal{F}$ and values $a_1, \ldots, a_k$ in Dom x. For location $\ell = f(a_1, \ldots, a_k)$, we sometimes write $x[\ell]$, instead of $f^x(a_1, \ldots, a_k)$ for the value that x assigns to ℓ. All terms in this paper are ground terms, that is, terms without variables.

We also assume that all elements of the domain are accessible via terms in initial states (or else the superfluous elements may be removed with no ill effect).

It is convenient to view each state as a collection of the graphs of its operations, given in the form of a set of location-value pairs, each written conventionally as $f(\bar{a}) \mapsto c$, for $\bar{a} \in$ Dom x, $c \in$ Dom x. Define the *update set* $\Delta(x)$ of state x as the changed points, $\tau(x) \setminus x$. When x is a terminal state and $\tau(x)$ is undefined, then we will indicate that by setting $\Delta(x) = \bot$.

The transition function of an algorithm must be describable in a finite fashion, so its description can only refer to finitely many locations in the state by means of finitely many terms over its vocabulary.

Postulate III (Effective Transitions)**.** *An algorithm with states $\mathcal{S}$ over vocabulary $\mathcal{F}$ determines a finite set T of* critical terms *over $\mathcal{F}$, such that states that agree on the values of the terms in T also share the same update sets. That is,*

$$\textit{if } x =_T y \textit{ then } \Delta(x) = \Delta(y) \ ,$$

for any two states $x, y \in \mathcal{S}$.

Here, $x =_T y$, for a set of terms T, means that $[\![t]\!]_x = [\![t]\!]_y$ for all $t \in T$.

Whenever we refer to algorithms below, we mean "sequential algorithms", satisfying the above postulates.

An algorithm A with states $\mathcal{S}$ *computes* a partial function $f : D^k \rightharpoonup D$ if there is a subset $\mathcal{I}$ of its initial states such that

1. The domain of each state in $\mathcal{I}$ is D.
2. There are k locations $\ell_1, \ldots, \ell_k$ such that $\{\langle x[\ell_1], \ldots, x[\ell_k]\rangle : x \in \mathcal{I}\} = D^k$.
3. All states in $\mathcal{I}$ agree on the values of all locations other than $\ell_1, \ldots, \ell_k$.
4. There is a location ℓ such that for all $a_0, \ldots, a_k \in D$, if $f(a_1, \ldots, a_k) = c$, then there is an initial state $x_0 \in \mathcal{I}$, with $x_0[\ell_j] = a_j$ $(j = 1, \ldots, k)$, initiating a terminating computation $x_0 \rightsquigarrow_\tau \cdots \rightsquigarrow_\tau x_n$, where $x_n \in \mathcal{O}$, such that $x_n[\ell] = c$.
5. Whenever $f(a_1, \ldots, a_k)$ diverges, there is an initial state $x_0 \in \mathcal{I}$, with $x_0[\ell_j] = a_j$ $(j = 1, \ldots, k)$, initiating an infinite computation $x_0 \rightsquigarrow_\tau x_1 \rightsquigarrow_\tau \cdots$.

A set of algorithms, all with the same domain, will be called a *model (of computation)*.

3 Effective Models

We describe now the three different approaches to effectiveness.

3.1 Distinguishing Algebras

Every state x induces a congruence on all terms under which terms are congruent whenever the state assigns them the same value:

$$s \simeq_x t \ \Leftrightarrow \ [\![s]\!]_x = [\![t]\!]_x \ .$$

Isomorphic states clearly induce the same congruence.

We will call a state "distinguishing" if its induced congruence is decidable (in the standard sense, via a Turing machine). This is the notion of effective state explored in [Rei08].

Definition 1 (Distinguishing Model)**.**

- *A state is* distinguishing *if its induced congruence is decidable.*
- *An algorithm is* distinguishing *if all its initial states are.*
- *A model is* distinguishing *if the congruence induced by every finite set of of initial states (across different algorithms) is decidable.*

The partial recursive programs are distinguishing by this definition.

3.2 Computable Algebras

We say that an algebra $\mathcal{A}$ over (a possibly infinite) vocabulary $\mathcal{F}$ with domain D *simulates* an algebra $\mathcal{B}$ over (a possibly infinite) vocabulary $\mathcal{G}$ with domain E if there exists an injective "encoding" $\rho : E \to D$ such that for every function $g : E^k \to E$ of $\mathcal{B}$ there is a function $f : D^k \to D$ of $\mathcal{A}$, such that $g = \rho^{-1} \circ f \circ \rho$.

Definition 2 (Computable Model)**.**

- *A state is* computable *if it is simulated by the recursive functions.*
- *An algorithm is* computable *if all its initial states are.*
- *A model is* computable *if all its algorithms are, via the same encoding.*

This is the standard notion of "computable algebra"; see [SHT95].
Clearly the partial recursive functions are computable by this definition.

3.3 Constructive Algebras

Let x be an algebra over vocabulary $\mathcal{F}$ and with domain D. A finite vocabulary $\mathcal{C} \subseteq \mathcal{F}$ *constructs* D if x assigns each value in D to exactly one term over C.

Definition 3 (Constructive Model)**.**

- *A state is* constructive *if it includes constructors for its domain, plus operations that are almost everywhere undefined, meaning that all but finitely-many locations have the same default value (say* undef*).*
- *An algorithm is* constructive *if its initial states are.*
- *A model is* constructive *if all its algorithms are, via the same constructors.*

Constructive algorithms can be bootstrapped: Any algebra over vocabulary $\mathcal{F}$ with domain D is constructive if $\mathcal{F}$ can be partitioned into $\mathcal{C} \uplus \mathcal{G}$ so that $\mathcal{C}$ constructs D and every $g \in \mathcal{G}$ has a constructive algorithm over $\mathcal{C}$ that computes it.

This is the approach advocated in [BD08].

3.4 The Moral

One can demonstrate the following:

Theorem 1. *A model of computation is computable if and only if it is constructive if and only if it is distinguishing.*

References

[BD08] Udi Boker and Nachum Dershowitz. The Church-Turing thesis over arbitrary domains. In Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich, editors, *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, volume 4800 of *Lecture Notes in Computer Science*, pages 199–229. Springer, 2008.

[DG08] Nachum Dershowitz and Yuri Gurevich. A natural axiomatization of computability and proof of Church's Thesis. *Bulletin of Symbolic Logic*, 14(3):299–350, 2008.

[Gur00] Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1:77–111, 2000.

[Mon60] Richard Montague. Towards a general theory of computability. *Synthese*, 12(4):429–438, 1960.

[Myh52] John Myhill. Some philosophical implications of mathematical logic. Three classes of ideas. *The Review of Metaphysics*, 6(2):165–198, 1952.

[Rei08] Wolfgang Reisig. The computable kernel of abstract state machines. *Theoretical Computer Science*, 409:126–136, 2008.

[Rog66] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1966.

[Sha82] Stewart Shapiro. Acceptable notation. *Notre Dame Journal of Formal Logic*, 23(1):14–20, 1982.

[SHT95] V. Stoltenberg-Hansen and J. V. Tucker. *Effective Algebra*, volume 4 of *Handbook of Logic in Computer Science*, chapter 4, pages 357–526. Oxford University Press, 1995.