THE LOGIC IN COMPUTER SCIENCE COLUMN[1]

by

Yuri GUREVICH[2]

The Sequential ASM Thesis

The thesis is that every sequential algorithm, on any level of abstraction, can be viewed as a sequential abstract state machine [Gurevich 1985, 1991]. Abstract state machines (ASMs) used to be called evolving algebras. The sequential ASM thesis and its extensions [Gurevich 1995] inspired diverse applications of ASMs. The early applications were driven, at least partially, by the desire to test the thesis. Different programming languages were the obvious challenges: Modula-2 [Gurevich and Morris 1987], Prolog [Börger 1990a, 1990b, 1992], C [Gurevich and Huggins 1992], etc. (A programming language L can be viewed as an algorithm that runs a given L program on given data.) From there, applications of (not necessarily sequential) ASMs spread into many directions; see the recent bibliography [Börger and Huggins 1998] and the ASM websites [ASM Michigan, ASM Paderborn]. So far, the accumulated experimental evidence seems to support the sequential thesis. There is also a speculative philosophical justification of the thesis. It was barely sketched in the literature, but it was discussed at much greater length in numerous lectures of mine. Here I attempt to write down some of those explanations. This article does not presuppose any familiarity with ASMs.

In Section 1, we discuss the need for a computation model that is more powerful and more universal than the standard computation models of theoretical computer science. In Section 2, we analyze sequential algorithms that are closed in the sense that they do not interact with their environments. Sequential abstract state machines are defined in Section 3. The sequential ASM thesis, restricted to closed algorithms, is explained, argued for and partially proved in Sections 3 and 4. In Section 5, the sequential ASM thesis is extended to algorithms interacting with their environments. Section 6 contains various remarks.

[1]Bulletin of European Association for Theoretical Computer Science, February 1999

[2]Microsoft Research, One Microsoft Way, Redmond, WA 98052-6399, USA, on a leave of absence from the University of Michigan.

1 The Problem

Quisani: I was surprised to find your name at the website of Microsoft Research. How do you like it there?

Author: I love it. One has here a unique opportunity to combine theory and applications.

Q: What would you like to apply?

A: Formal methods, in particular ASMs, abstract state machines.

Q: What is so special about ASMs? What distinguishes them in the vast zoo of formal methods?

A: I like to think of ASMs as a bridge between the computation models of theoretical computer science and real-world problems of specification and verification.

Q: Do you view ASMs as a computation model?

A: Yes, I do.

Q: Does one need another computation model? I think Turing machines serve the purpose well.

A: The question is, what is the purpose exactly? What purpose are you talking about?

Q: Well, why did Turing introduce his machines on the first place? Did he try to formalize the notion of algorithm?

A: I don't think so. Both Church and Turing were interested in the classical decision problem: Find an algorithm that decides whether a given first-order formula is valid. It was one of the most intriguing logic problems of the 1930s. Hilbert called it the main problem of mathematical logic. Church and Turing gave different but equivalent formalizations of the notion of computable function and used their formalizations to show that the classical decision problem is unsolvable [Church 1936, Turing 1936]. In particular, Turing put forward a thesis that a function is computable if and only if it is computable by a Turing machine. He proved that no Turing machine computes the validity of first-order formulas. By the thesis, validity is not computable.

Q: Maybe the classical decision problem was only a pretext, and their real purpose was to capture the notion of computable function.

A: Maybe. I don't know.

Q: By the way, did they really capture the notion of computable function? A Turing computable function may not be computable in any practical sense.

A: Indeed Church-Turing computability is a liberal notion, but it gives a meaningful upper bound on the class of computable functions.

Q: Why is capturing the notion of computable function not the same as capturing the notion of algorithm?

A: There is more to an algorithm than the function it computes.

Q: A computable function is a function computable by an algorithm. How could Church and Turing speak about arbitrary computable functions without speaking about arbitrary algorithms?

A: They did speak about arbitrary algorithms. In fact, Turing's informal proof of his thesis justifies a somewhat stronger thesis: every algorithm can be simulated by a Turing machine.

Q: There you go. Turing did capture the notion of algorithm.

A: Not really. There may be a huge difference between an algorithm and the simulating Turing machine. The poor Turing machine may work long and hard, with its head creeping painfully slowly back and forth on that long narrow tape, in order to simulate just one step of the given algorithm.

Q: Now you are talking complexity. I thought that experts agree that any algorithm can be simulated by a Turing machine with only polynomial slowdown.

A: True, but polynomial time simulation may not be good enough. Think about practical algorithms.

Q: What kind of simulations would you like?

A: Step-for-step simulations.

Q: I recall now our conversation [Gurevich 1988c] involving Kolmogorov-Uspenski machines [Kolmogorov and Uspenski 1958] and Schönhage machines [Schönhage 1980]. A question has been bothering me. You spoke about the KU thesis — any algorithm can be simulated step-for-step by a KU machine, and about a similar thesis for Schönhage machines. However, it was unknown whether Schönhage machines could be simulated step-for-step by KU machines. Given that, how could you argue for the KU thesis?

A: We are getting closer to the heart of the matter. The difference between the two theses is in the levels of abstraction at which they consider algorithms. States of a KU machine reflect the physical world. In a finite-dimensional space, a physical representation of a bit may have only so many neighboring physical representations of bits. Accordingly, the memory of a KU machine is a graph of bounded degree. On the other hand, the memory of a Schönhage machine is a directed graph where only out-degree is bounded; there is no bound on in-degree. This is natural from the point of view of pointer computations on our computers, but it is a higher abstraction level.

Q: I guess the random access machines of [Cook and Reckhow 1973] reflect an even higher abstraction level?

A: Yes. The problem arises whether there is a machine model such that any algorithm, however abstract, can be simulated step-for-step by a machine of that model. Let me call such a model universal with respect to algorithms. Turing's machine is universal with respect to computable functions, but not with respect to algorithms.

Simulating algorithms on their natural abstraction levels, without implementing them, involves interesting issues. Consider for example algorithms that work with graphs. The conventional computation models [Savage 1998] require a string representation of the given graph or something like that even in those cases when, in principle, the algorithm is independent of the graph representation. An algorithm-universal computation model would allow one to program such algorithms in a way that is independent of graph representation.

Q: I see nothing wrong in using a data representation appropriate to the task and taking advantage of it. Any computer will require some kind of data representation.

A: It may be necessary to have the algorithm in a form that is independent of data representation. The problem is especially acute in databases. The same database may have different representations, but the meaning of the data should not change. Database query languages are supposed to reflect only representation-independent properties.

Q: Can an algorithm be truly independent of data representation? After all it deals with mathematical abstractions, not with real world objects. Do you see what I mean?

A: You are right. A graph algorithm deals with graphs and thus the raw data is somewhat structured already by the use of graphs. Similarly, the choice of database model structures data to a great extent. I should have been more careful. I meant to say this. An algorithm-universal computation model allows one to program algorithms on their natural abstraction levels, without imposing additional representation of data. For example, if your algorithm is supposed to deal with graphs, then the computation model allows you to deal with graphs without forcing you to adjacency matrices or any other kind of representation of graphs.

Q: Now I understand your intention.

A: An important use of an algorithm-universal computation model, if it is simple and clean, is to give operational semantics to computer systems.

Q: Don't count your chickens before they hatch. I don't believe there is an algorithm-universal computation model.

2 Sequential Algorithms

Q: The world of algorithms is too diverse and there are too many possible levels of abstraction. How can one computation model reflect them all? Consider, for example, the following models of random access machine: the successor RAM, whose only operation is $x \mapsto x + 1$; the addition RAM; the RAM with addition and multiplication; the RAM with addition, multiplication and division. Does your algorithm-universal model do multiplication? If so, its abstraction level is too high for the first two RAM models. If not, its abstraction level is too low for the last two RAM models.

A: It is easy to unify your RAM models. Introduce a RAM model with a parameter that is a set of arithmetical operations.

Q: And how will you unify the RAM models with the Turing machine and with e.g. the C programming language? Furthermore, I was thinking about sequential algorithms till now, but there are also parallel, distributed, real-time algorithms. How broadly do you want to understand the notion of algorithm?

A: Let's restrict this discussion to sequential algorithms. We can discuss more general algorithms some other time.

Q: What are sequential algorithms exactly? Explain your intuition.

A: I'll try. Let me start with some obvious points.

Point one: a sequential algorithm A, like any other algorithm, is given by a finite text, the program of the algorithm. Typically it is a program in some programming language.

Point two: there are two collections associated with A, the collection of all states of A and the collection of initial states of A. The second is a subcollection of the first.

The third point is that A works in sequential time.

Q: What do you mean by that?

A: There is a state-transformation φ associated with A, the transformation induced by executing one step of A. If S is a state of A, then φS is also a state of A.

Q: What if S is a final state of A?

A: Stipulate that $\varphi S = S$ in that case.

The transition system gives rise to the notion of run. Call an algorithm *closed* if it does not interact with its environment. For simplicity, let me restrict attention to the case when A is closed. We can discuss the more general case later. In the restricted case, a *run* (or *computation*) of A can be defined as a sequence

$$S_0, S_1, S_2, \ldots$$

where S_0 is an initial state and every $S_{i+1} = \varphi(S_i)$.

Q: Can a run be infinite?

A: Yes, there are many useful algorithms that are not supposed to terminate.

Q: What if a computation encounters an error?

A: Errors are in the eye of the beholder. An error state is still a state.

Q: Do you assume that every state is reachable, that is, appears in some run?

A: No, think about *a priori* states of the algorithm. Often, one describes the states of the algorithm without knowing which of them are reachable. Recall Turing machines, but distinguish between the states of the finite control and the states (or configurations) of the machine itself; one halting state of the finite control gives rise to many halting states of the machine. It is undecidable whether a given Turing machines reaches a halting state. Nevertheless, halting states are legitimate states.

Q: OK, I get the third point.

A: The fourth point is that a sequential algorithm has only one execution thread. In other words, there is only one executing agent. This distinguishes sequential algorithms from multi-agent (= distributed) algorithms.

Q: There are programming languages where matrix multiplication can be done in one step. In reality of course matrix multiplication may require a lot of work. It looks like the agent that pretends to do matrix multiplication in one step needs help of another agent or agents.

A: On the higher abstraction level where matrix multiplication is done in one step, we ignore how multiplication is actually performed. All we care is that our agent does it in one step.

Q: Are you through with your description of sequential algorithms?

A: No, not yet. There is also a problem, how much work can a sequential algorithm do in one step? For example, let us consider an algorithm which, given a graph (V, E) with distinguished vertices Source, Target, checks whether Target is reachable from Source. An auxiliary relation R (an allusion to "reachable") is initially empty. The algorithm iterates the following step:

```
if $\neg R$(Source) then $R$(Source) $:=$ true
elseif $\exists x \exists y(R(x) \wedge E(x,y) \wedge \neg R(y))$ then
   do for all $x, y$ with $R(x) \wedge E(x,y) \wedge \neg R(y)$
      $R(y) := true$
   enddo
else Output := $R$(Target)
endif
```

Is this algorithm sequential?

Q: Definitely not. This algorithm is highly parallel, not sequential. It performs a number of actions at the same time.

A: A sequential algorithm also can perform several actions a time. For example, at one step, a Turing machine may change its control state, print a symbol at the current tape cell and move its head. Similarly, a programming language interpreter may perform an arithmetic operation and change the program counter; it may also change the procedure stack at the same occasion.

Q: There is a difference in the amount of parallelism. A sequential algorithm may perform only so many actions a time. I guess your fifth point is that the number of actions performed by a sequential algorithm at any step is bounded; is that it?

A: We are getting there. What do you mean by an action?

Q: Something that changes the state.

A: Consider the following algorithm which checks whether the given graph (V, E) has isolated points.

```
if $\forall x \exists y E(x,y)$ then Output := false
else Output := true
```

Q: I see. The algorithm does not change much but is massively parallel nonetheless. Obviously we should take into account actions that inspect the state without changing it. I am not sure how to count such inspection actions. Coming to think of it, I am not sure how to count actions that do change the state. One may claim that every Turing machine performs only one action a time, but that action have several parts. The number of parts is in the eye of the beholder. You counted three parts. I can count four parts by requiring that the old symbol is erased before the new symbol is printed. Also, the new symbol may be composed, e.g. "12". Printing a composed symbol can be a composed action all by itself.

A: Now I am ready to formulate my fifth and last point. The amount of work that a sequential algorithm does at any step is bounded.

Q: But how do you define work? How do you measure work?

A: Good questions.

3 Derivation of Sequential ASMs

Q: I recall now that we spoke once about evolving algebras[3]. Have they been renamed to abstract state machines?

A: Yes.

Q: Do you claim that the ASM model is algorithm-universal?

A: Well, this is the ASM thesis.

Q: The difference between a thesis and a theorem is, I guess, that a theorem has a proof.

A: A theorem relates mathematical objects; its proof is a mathematical proof. A thesis relates mathematical and non-mathematical objects. It cannot have a mathematical proof, but it may be affirmed or negated by experimentation. It may also have some kind of speculative justification. For example, Turing gave a convincing speculative proof of his thesis.

[3]There was no previous discussion on the subject in this column. Quisani has in mind the tutorial [Gurevich 1991].

Q: Speculate about your thesis.

A: OK. I will formulate portions of the thesis as claims (or subtheses if you will). Let A range over sequential algorithms.

Claim 1 *Every state of A can be viewed as a first-order structure of a finite vocabulary. Furthermore, the collection of states is closed under isomorphisms.*

Q: What do you mean by "can be viewed as a first-order structure"?

A: Can be adequately described as a first-order structure and therefore can be identified with that structure.

Q: Why is the collection of states closed under isomorphism?

A: We abstract from any particular presentation of a state and thus only the isomorphism type of the state matters.

Q: Are you thinking about highly abstract algorithms?

A: No, the abstraction level does not have to be high. It may be for example the level of assembly language. But the abstraction level should be fixed. If some details of the presentation of the state still matter, this means that the abstraction level has not been chosen correctly. Details that matter should be incorporated into the state.

Q: I need to see examples. Describe the states of a Turing machine as structures.

A: A state of a Turing machine can be formalized as follows. The basic set of the structure includes the union

$$\text{Control} \cup \text{Alphabet} \cup \text{Tape}$$

of three disjoint sets. Formally Control, Alphabet and Tape are unary relations, but it is convenient to view them as separate universes (or subuniverses).

- Control is (or represents) the set of states of the finite control. Each element of Control is distinguished, that is, has a name in the language. These names are nullary function names. In addition, there is a dynamic distinguished element CurrentControl that "lives" in Control but may change its value from one state of the machine to another. Speaking more accurately, we have the nullary function name CurrentControl whose value may change from one state to another.

- Alphabet is the tape alphabet. Each element of Alphabet is distinguished as well. One of this elements is called Blank.

- Tape can be taken to be the set of natural numbers (representing tape cells) complete with 0 as a distinguished element and with the unary operations Successor and Predecessor (with Predecessor(0) = 0 for example). There is also a dynamic nullary function name Head that takes values in Tape.

Finally, we have a unary function

$$\text{Content} : \text{Tape} \to \text{Alphabet}$$

which assigns Blank to all but finitely many cells.

You can use this notation to write Turing instructions. For example,

```
if CurrentControl = $q_1$ and Content(Head) = $\sigma_1$ then
   do in-parallel
      CurrentControl := $q_2$
      Content(Head) := $\sigma_2$
      Head := Successor(Head)
   enddo
endif
```

Q: Give me another example. How about the Euclidean algorithm?

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

```
if $b = 0$ then $d := a$
elseif $b = 1$ then $d := 1$
else
   do in-parallel
      $a := b$
      $b := a \ mod \ b$
   enddo
endif
```

Now it is pretty obvious what the states are. The basic set includes the set of natural numbers which comes with $0, 1$ as distinguished elements and with the binary operation *mod*. In addition, there are three dynamic distinguished elements a, b and d.

Q: Why do you think that the notion of first-order structure is sufficient to describe the states of arbitrary algorithms?

A: Logic experience shows that any kind of static mathematical situation can be adequately described as a first-order structure.

Q: Then why do logicians use other kinds of structures, e.g. second-order structures? How would you deal, for example, with a graph algorithm that manipulates not only vertices but also sets of vertices?

A: Second-order structures can be seen as special first-order structures. The second-order graph structure can be viewed as a two-sorted first-order structure where elements of the first sort are vertices and elements of the second sort are vertex sets; in addition to the edge relation on the first sort, there is also a cross-sort binary relation $\epsilon(x, y)$ expressing that a vertex x belongs to a vertex set y. Formally speaking, sorts are just unary relations.

Q: Then who needs second-order logic?

A: There is more to second-order logic than just the definition of second-order structures. There are various second-order calculi. Presently we are concerned only with structures, not with the ways to reason about them.

Q: In the case of Turing machines, you set Predecessor(0) = 0. It would be more natural to declare that Predecessor(0) is undefined, but you cannot do that. The operations of a first-order structure are everywhere defined.

A: Indeed, I tried to simplify things a little. In reality, we restrict attention to structures satisfying the following requirements.

- Every structure contains a distinguished element called `undef` (or `default`).

This allows us to deal with partial functions.

- Every structure contains distinguished elements `true` and `false`.

This allows us to view relations as special functions.

- Every structure contains the equality relation and the standard Boolean connectives which behave in the expected way over $\{\texttt{true}, \texttt{false}\}$.

The states described above should be augmented to accommodate these obligatory functions.

Q: According to Claim 1, a computation is a sequence of various structures. What does it buy you?

A: This question brings us to the next claim which strengthens Claim 1. Recall that A is an arbitrary sequential algorithm.

Claim 2 *All states of A are structures of the same finite vocabulary. In particular, the vocabulary does not change during the computation.*

Q: Why doesn't the vocabulary change during the computation? I can imagine an algorithm that needs more and more functions or relations as it runs. For example, an algorithm that colors natural numbers may need more and more colors. It is natural to think of colors as disjoint unary relations. Thus we have a growing collection of unary relations.

A: In a proper formalization, the vocabulary reflects only truly invariant features of the algorithm rather than details of a particular state. In your example, how does the coloring algorithm refer to various colors? It may use for example an array of colors. Mathematically this gives rise to a *binary* relation $C(i, x)$ where the i^{th} color is $\{x : C(i, x)\}$. In general, if an algorithm needs more and more j-ary functions of some kind, it may really deal with a $(j + 1)$-ary function. Alternatively, it may be appropriate to treat these j-ary functions as elements of a special universe.

Q: There are so-called "non-von-Neumann" algorithms which change their programs during the computation.

A: I think that, for such an algorithm, the so-called program is just a part of data. The real program changes that part of data, and the real program does not change.

Q: According to Claim 2, a computation is not just a sequence of structures, but a sequence of structures of the same vocabulary. What does it tell us?

A: That the vocabulary does not change during the computation. Think about a computation as an evolution of the initial state. The vocabulary does not change during that evolution.

Q: Does the basic set change during the evolution?

A: Without loss of generality, we may assume that it doesn't.

Q: I do not understand this. I think the basic set may change. For example, a graph algorithm may require that new vertices are added to the graph.

A: Sure, but where do the new vertices come from? The initial state can have an infinite reserve where potential vertices "live". In general, it is convenient to assume that the initial state has enough material for the evolution and thus the basic set does not change during the evolution. Let me formulate this as another claim. Again, let φ be the state transformation performed by our algorithm A in one step.

Claim 3 *The one-step transformation φ does not change the basic set of any state.*

Q: I have to think about that.

A: Let us continue.

Claim 4 *If ι is an isomorphism from a state S to another state S', then it is also an isomorphism from φS to $\varphi S'$.*

Q: Why?

A: View S' as just another representation of S. An element a of S is represented by an element $\iota(a)$ of S'. But the representation of a state should not matter. Accordingly φ operates in the same way at both structures.

Q: So what does change during the evolution?

A: Some basic functions of the structure may change. In the case of Turing machines, the functions CurrentState, Head and Content change. In the case of the Euclidean algorithm, the nullary functions a, b and d change.

It is convenient to think about a state S as a memory of a kind. If f is a j-ary function name and $\bar{a}$ is a j-tuple of elements of (the basic set of) S, then the pair $(f, \bar{a})$ is a *location*. The content of that location is the value $\text{Value}_S(f(\bar{a}))$ of f at $\bar{a}$ in S. Since the one-step transformation φ changes neither the vocabulary nor the basic set of a given set S, the resulting state φS has exactly the same locations. Of course, some locations may have different contents in φS.

One important observation is that the change imposed by φ on S splits into atomic, indivisible changes. If ℓ is a location of a state S and b is an element of S, call the pair (ℓ, b) an *update* of S. Let $\text{Content}_S(\ell)$ denote the content of ℓ in S. The update (ℓ, b) is *trivial* if $b = \text{Content}_S(\ell)$. A non-trivial update represents an atomic change. The full change imposed by φ on S is the set $\text{UpdateSet}_\varphi(S)$ of nontrivial updates

$$(\ell, \text{Content}_{\varphi S}(\ell))$$

over S. To reflect atomic changes, the sequential ASM programming language provides special *update rules*.

Recall the recursive definition of ground terms. For brevity, I will omit the adjective ground.

- Every nullary function name f is a term.
- If f is a function name of arity $j > 0$ and $t_1, \ldots, t_j$ are terms, then $f(t_1, \ldots, t_j)$ is a term.

Given a structure S consider only terms t such that every function name in t belongs to the vocabulary of S. The value $\text{Value}_S(t)$ of t in S is defined in the obvious way.

An update rule has the form

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

where j is the arity of f and $t_0, \ldots, t_j$ are terms. To fire the rule at a structure S where the terms $t_0, \ldots, t_j$ evaluate to elements $a_0, \ldots, a_j$ respectively, implement update (ℓ, a_0) over S where $\ell = (f, (a_1, \ldots, a_j))$. Since relations are special functions, update rules allow us to update relations as well.

Q: I do not see how update rules allow you to change arbitrary locations. An update rule deals only with a location $(f, (a_1, \ldots, a_j))$ where each element a_i is specified by a term. The new value that it puts into the location has to be specified by a term as well.

A: You are right. Let me call an element a of a state S *accessible* if there exists a term t with $\text{Value}_S(t) = a$. A tuple $(a_1, \ldots, a_j)$ is *accessible* if every element a_i is accessible. A location $(f, \bar{a})$ is *accessible* if the tuple $\bar{a}$ is accessible. Indeed, the update rules deal only with accessible locations and accessible values. But this is by design. Let again A be an arbitrary sequential algorithm and φ be the one-step state transformation of A.

Claim 5 *φ cannot change the content of an inaccessible location or put an inaccessible element into an accessible location.*

Q: Why do you believe that?

A: Imagine yourself being an executor of φ at a state S. Since you cannot take advantage of a particular representation of the elements, the only way to access elements of S is to use the basic functions of S. Essentially you use S as an oracle. If you can name a location ℓ then S will give you $\text{Content}_S(\ell)$. You can find out the value $\text{Value}_S(f)$ of any nullary function name f. If f is a function name of arity j and you can name elements $a_0, a_1, \ldots, a_j$, then you can name element $f(a_1, \ldots, a_j)$ and you can put a_0 into location $(f, (a_1, \ldots, a_j))$. That is all that you can do.

Q: I wonder if there are other means to point to an element without using any representation of it.

A: What do you mean by pointing to an element?

Q: Consider for example an algorithm that works with directed graphs with a distinguished vertex Star, so that Star is the only accessible vertex. If a is the unique vertex with an edge to Star, then a is uniquely identified by its description, even though it may be inaccessible.

A: In order to determine that a is the unique vertex with an edge to Star, the executor must check all vertices. So going from Star to a is not bounded work.

Q: It may be bounded work on a higher abstraction level. Maybe the code of the algorithm contains some procedure which, given a vertex x, checks whether it has exactly one incoming edge and, if yes, produces the source of that edge. But I do not want to know anything about that procedure. I'd like to be on the abstraction level where, by some kind of magic, we can go from any given vertex x with exactly one incoming edge, to the source of that edge in one step.

A: To abstract from the procedure, introduce a new basic function Source(x). If x has exactly one incoming edge (y, x), then Source(x) $= y$; otherwise Source(x) $=$ `undef`.

Let me add that parallelism and nondeterminism allow one to deal with term-unreachable elements and locations, but here we consider only deterministic sequential algorithms.

Q: What else do you have in the ASM programming language?

A: Since an algorithm can perform several changes at a time, the ASM programming language provides rules of the form

```
do in-parallel
    $R_1$
    $R_2$
    ⋮
    $R_k$
enddo
```

where $R_1, \ldots, R_k$ are constituent rules constructed earlier. The keywords "do in-parallel" and "enddo" are often omitted. To fire a do-in-parallel rule, fire the constituent rules simultaneously. In other words, the update set generated by the do-in-parallel rule is the union of the update sets generated by the constituent rules.

Q: What if R_1 and R_2 contradict each other? For example, how do you execute the following?

```
do in-parallel
    Head := Predecessor(Head)
    Head := Successor(Head)
enddo
```

A: You cannot execute that. Formally speaking, the computation halts.

Q: Why not choose one of the two possible actions and execute it?

A: This will introduce nondeterminism. We are talking about deterministic algorithms presently.

Q: But nondeterminism may be convenient.

A: Sure. The notion of nondeterministic ASM is one of several generalizations of the notion of deterministic sequential ASM [Gurevich 1995].

Q: The keyword "do in-parallel" appeared earlier in our discussion. I guess you have been using the ASM programming language all along.

A: I have. And thus you have also seen conditional rules. If φ is a Boolean-valued term (essentially a quantifier-free first-order formula) and R_1, R_2 are rules, then

```
if $\varphi$ then
   $R_1$
else
   $R_2$
endif
```

is a rule as well. The `elseif` that you saw in the Euclidean algorithm resulted from nesting two conditional rules. It is obvious how to fire a conditional rule.

Q: What else is there in the deterministic sequential ASM programming language?

A: There are no other ways to generate rules.

Q: And what is a program?

A: A program is just a rule.

Q: And what is an ASM?

A: An ASM M is given by a vocabulary, a program of that vocabulary (so that all function names of the program occur in the vocabulary), a collection of states of M, and a collection of initial states of M. It is supposed of course that every state of M is a state of the vocabulary of M, and every initial state of M is a state of M, and firing the program of M at any state of M results in a state of M. Intentionally, M is an algorithm. Notice that the program describes only one step of the algorithm and is supposed to be fired over and over again.

We concentrate on the dynamic part and impose no *a priori* restriction on how to describe the collection of states and the collection of initial states.

Q: There is no sequential composition of rules? I mean something like

$$R_1 \;\; ; \;\; R_2$$

First do R_1 and then do R_2.

A: There is no sequential composition. Recall that the program of an ASM describes only one step of the algorithm. It is an unusual programming style; it forces you to stay on one level of abstraction. The sequential composition allows you to compose trees, and forests, and bushes, and grass, which makes reasoning harder.

Q: Can ASMs handle recursion?

A: Yes, but we view recursive computations being implicitly multi-agent [Gurevich and Spielmann 1997].

Q: Your programming language seems poor to me.

A: Nevertheless, the full sequential ASM thesis is that every bounded-work sequential algorithm can be adequately represented by an appropriate ASM. Actually it can be proved given the claims above and given an appropriate definition of bounded work.

Q: I would like to see how you do that.

4 Bounded Work

A: Again, let A range over sequential algorithms, and let φ be the state transformation induced by one step of A. By Claim 2, the states of A are structures of the same finite vocabulary Υ. Recall the stipulation that $\varphi S = S$ for every final state of A, so that φ is defined at every state of A.

Call φ a *bounded-work transformation* if there exists a finite set T of terms such that $\text{UpdateSet}_\varphi(S_1) = \text{UpdateSet}_\varphi(S_2)$ whenever $\text{Value}_{S_1}(t) = \text{Value}_{S_2}(t)$ for all $t \in T$.

Theorem 1 *If φ is bounded-work, then there exists an ASM program Π such that Vocabulary$(\Pi) \subseteq \Upsilon$ and the one-step transformation ψ associated with Π coincides with φ on all states of A. In other words,*

$$\mathit{UpdateSet}_\varphi(S) = \mathit{UpdateSet}_\psi(S)$$

for all states S of A.

Proof Let T witness that φ is bounded-work. Without loss of generality, T is closed under subterms. Terms in T will be called *critical.* For every state S, the elements $\mathrm{Value}_S(t)$ with $t \in T$ will be called *critical elements* of S.

Lemma 1 *If $u = (\ell, a_0) \in \mathit{UpdateSet}_\varphi(S)$ and $\ell = (f, (a_1, \ldots, a_j))$, then the elements $a_0, \ldots, a_j$ are critical.*

It follows that there is a bound on the size of $\mathrm{UpdateSet}_\varphi(S)$ that depends on φ but not on S.

Proof of Lemma 1. By contradiction, assume that some a_i is not critical. Let b_i be a fresh element and let S' be the state isomorphic to S which is obtained from S by replacing a_i with b_i. It is easy to see that, for every critical term t, $\mathrm{Value}_S(t) = \mathrm{Value}_{S'}(t)$. It follows that $\mathrm{UpdateSet}_\varphi(S) = \mathrm{UpdateSet}_\varphi(S')$. Therefore the update u belongs to $\mathrm{UpdateSet}_\varphi(S')$ but this is impossible because u involves element a_i which isn't an element of S'. $\square$

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

$$E_S(t_1, t_2) :\iff \mathrm{Value}_S(t_1) = \mathrm{Value}_S(t_2)$$

over critical terms. Call states S, S' *similar* if $E_S = E_{S'}$. Notice that, since there are only finitely many critical terms, there are only finitely many possible equivalence relations E_S. In other words, there is a finite collection $\{S_1, \ldots S_m\}$ of states such that every state is similar to one of the states S_i.

Lemma 2 *For every state S of A, there exists an ASM program Π_S such that $\mathit{Vocabulary}(\Pi_S) \subseteq \Upsilon$ and the one-step transformation ψ_S associated with Π_S coincides with φ on all states of A similar to S.*

Proof of Lemma 2. Consider any update $u = (\ell, a_0) \in \mathrm{UpdateSet}_\varphi(S)$ where $\ell = (f, (a_1, \ldots, a_j))$. By Lemma 1, there exist critical terms $t_0, \ldots, t_j$ with $\mathrm{Value}_S(t_i) = a_i$, so that u can be generated by the following update rule R_u:

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

We show that, over any state S' similar to S, the update

$$(\ell', \mathrm{Value}_{S'}(t_0)) \quad \text{where} \quad \ell' = (f, (\mathrm{Value}_{S'}(t_1), \ldots, \mathrm{Value}_{S'}(t_j)))$$

generated by R_u belongs to $\mathrm{UpdateSet}_\varphi(S')$. Without loss of generality, S and S' have no common elements. Change S' by replacing every critical element $\mathrm{Value}_{S'}(t)$

with the corresponding critical element $\text{Value}_S(t)$ in S. The resulting structure S'' is isomorphic to S'; by Claim 1, S'' is a state of A. Further, S'' has the same values for the critical terms as S does. Thus $\text{UpdateSet}_\varphi(S) = \text{UpdateSet}_\varphi(S'')$ and therefore R_u generates one of the updates in $\text{UpdateSet}_\varphi(S'')$. Since S' and S'' are isomorphic, R_u generates one of the updates in $\text{UpdateSet}_\varphi(S')$.

The desired Π_S is the do-in-parallel block of all rules R_u where u ranges over $\text{UpdateSet}_\varphi(S)$. □

Q: Something bothers me in that proof. Suppose that A itself is given by a block Π of update rules and that one of the update rules, let me call it R, is $f(s) := t$ where the terms s, t are critical but the term $f(s)$ is not. Suppose further that $f(s)$ and t have the same values in S but different values in S'. Since R does nothing over S, Π_S will overlook R. However, R generates a nontrivial update over S', and so Π_S will not simulate Π correctly over S'.

A: That scenario seems plausible but it is contradictory. Since S'' is isomorphic to S', R generates a nontrivial update over S''. But $\text{UpdateSet}_\varphi(S) = \text{UpdateSet}_\varphi(S'')$ and thus, contrary to one of your assumptions, R generates a nontrivial update over S.

Now let's finish the proof of the theorem. Notice that, for every S, there exists a Boolean-valued term δ_S that takes value `true` in a structure S' if and only if S' is similar to S. Let Π_S be as in Lemma 2. Consider a maximal collection $S_1, \ldots, S_m$ of states of A such that every state is similar to one of the states S_i. The desired ASM program Π is the program

```
if       $\delta_{S_1}$  then   $\Pi_{S_1}$
elseif   $\delta_{S_2}$  then   $\Pi_{S_2}$
...
elseif   $\delta_{S_m}$  then   $\Pi_{S_m}$
endif
```

That completes the proof of the Theorem. □

5 Sequential Algorithms and their Environments

Q: Until now, we have discussed sequential algorithms that do not interact with the outside world, closed algorithms. In reality, many sequential algorithms do interact with their environments. How do you incorporate the environment into the picture?

A: Intuitively, an active environment is an outside agent.

Q: In reality there may be several outside agents influencing our algorithm.

A: True. In the distributed paradigm, we deal explicitly with multiple agents [Gurevich 1995]. In the sequential paradigm, agents are implicit, and the environment accounts for all outside agents. Typically, we do not have a program for the active environment, but we may know some restrictions, or integrity constraints, on its actions. Consider for example the following version of the Euclidean algorithm which repeatedly computes the greatest common divisor. For brevity, I omit the keywords "do in-parallel" and "enddo"; this is a common practice in ASM applications.

```
if Mode = Initial then
   a := Input1, b := Input2, Mode := Compute
endif

if Mode = Compute then
   if $b = 0$ then
      $d := a$, Mode := Initial
   elseif $b = 1$ then
      $d := 1$, Mode := Initial
   else
      $a := b$, $b := a \; mod \; b$
   endif
endif
```

Here Input1 and Input2 are changed by the environment. It is assumed though that they are natural numbers.

Q: What else can they be? I thought that the basic set of the Euclidean algorithm contains only natural numbers.

A: Recall that every state contains (elements called) `true, false, undef`. The definition of states of the generalized Euclidean algorithm may allow you to have other elements, even though those additional elements are irrelevant as far as our algorithm is concerned.

Q: Do you want to reconsider any of the five claims in Section 3?

A: No, the five claims can be justified as before.

Q: But the situation is quite different. A sequential algorithm can input and output data as it runs.

A: We can abstract from the output phenomenon. As far as our algorithm is concerned, the output is miraculously taken away. To account for the input phenomenon, change the notion of run. Again, let A be a sequential algorithm and φ be the state transformation induced by executing one step of A. If A is not influenced by the environment, then a run of A is a sequence

$$S_0, S_1, S_2, \ldots$$

of states where S_0 is an initial state and every $S_{i+1} = \varphi S_i$. In general, a run is a sequence

$$S_0, S_1, S_1', S_2, S_2', \ldots$$

of states where S_0 is an initial state, $S_1 = \varphi S_0$, S_1' is obtained from S_1 by an action of the environment, $S_2 = \varphi S_1'$, S_2' is obtained from S_2 by an action of the environment, and so on.

Q: Do you mean that there is a particular state transformation ψ such that $S_i' = \psi S_i$?

A: If particular means deterministic, then the answer is no. It may be possible that S_i coincides with S_j, but S_i' differs from S_j'. Consider for example a run of the generalized Euclidean algorithm where the first pair of inputs equals the third pair of inputs. The fourth pair of inputs can be different from the second pair.

A related phenomenon is that, in the case of active environment, a computation that comes across contradicting updates may not halt forever. You may have a scenario where the update set of, say, S_7' is contradictory, so that $S_8 = S_7'$, but — due to the active environment — S_8' differs from S_8, and the update set of S_8' is consistent.

Q: Or the update set of S_0 is contradictory, $S_1 = S_0$ and the update set of S_1' is consistent.

A: Right.

Q: It is strange that the algorithm and the environment take turns to act. In the case of the generalized Euclidean algorithm for example, the environment may prepare new Input1 and Input2 at the time that the algorithm works on the old Input1 and Input2.

A: The definition of run reflects the point of view of the executor of our algorithm. He will not see the new inputs until he finishes with the old ones. As far as he is concerned, the new inputs could appear after he finishes with the old ones. Different agents may see the same computation very differently.

Q: I guess Theorem 1 remains valid.

A: Yes. There is no need to extend the ASM programming language.

Q: Isn't the new definition of run too general? S_i' may be any state, independently of what S_i is.

A: I think the generality is needed, but indeed, in applications, the environment usually satisfies stringent restrictions. For example, it may change only particular functions or particular locations. Typically, it changes only a bounded number of locations at a time. Often, as in the case of the generalized Euclidean algorithm, there are several functions $f_1, \ldots, f_k$ under complete control of the environment, but the environment cannot change any other function. In such a case, it may be convenient to think that the algorithm computes S_{i+1} directly from S_i querying the environment about the values of *external* (or *monitored*) functions f_i.

Q: What if A reaches some state S_i and then queries the environment, but the environment does not reply, so that A is stuck forever in S_i. I can imagine for example that the generalized Euclidean algorithm waits for me to provide Input1 and Input2, but I have no intention to do so.

A: Then S_i is the last state of the run.

Q: A practical algorithm may use a timer and thus wait only so long.

A: True. But the notion of real-time algorithm is one of those generalizations that we will have to discuss some other time.

Q: Let me come back to Claim 3 according to which the basic set does not change during the computation. A program that runs on your computer may require additional space from time to time. It may issue a NEW command or something like that and get more space.

A: On a higher abstraction level, we may not see the space. We will see only new elements.

Q: I understand that. The problem is that our algorithm does not do anything to acquire the new elements except for asking for them. You've mentioned that the initial state of a computation may have a reserve of potential elements. How is one supposed to get elements from the reserve?

A: What do you think?

Q: I don't know. You may assume that the reserve is organized as natural numbers with a distinguished initial element and a unary successor function. When you need another element, you remove that initial element from the reserve and make its successor initial. This is not much work, but this is something that the original algorithm does not do.

A: Who does this work?

Q: I see where you are going. It is the task of the environment to provide new elements.

A: Right. Intuitively, the reserve is a naked set. We may use one of those external functions to fish out an element from the reserve. It seemed convenient to me to pretend nevertheless that our algorithm does the work. To this end, the ASM programming language provides special `create` (or `import`) rules

```
create x
   $R(x)$
endcreate
```

This `create` reflects a request to the environment for more resources. The environment obliges and provides a new element. Typically a create rule is used to extend some particular universe U and then it is abbreviated as `extend` U. For example, consider a Turing machine with finite tape where Last is the rightmost cell. An instruction may involve an extension of the tape:

```
extend Tape with $x$
   do in-parallel
      Successor(Last) := $x$
      Predecessor($x$) := Last
      Last := $x$
   enddo
endextend
```

Q: I guess Successor(x) = Predecessor(x) = `undef` for the reserve elements.

A: Correct. The reserve elements have no meaningful relationships except for the equality/inequality relationships.

Q: If you use `create` rules then your terms are not ground anymore.

A: Right. Some definitions have to be refined. For example, a program is now a rule without free variables.

Q: If you don't fish the new elements from the reserve but bring them from outside, then you have to extend all your functions. Of course, the extension is almost everywhere trivial, but still it involves unbounded work.

A: We do not require that the active environment is bounded-work. On the other hand, this discussion is supposed to be bounded-work. Let's finish here. It was a pleasure. See you around.

Q: Thank you. Bye.

6 Remarks

Now that my inquisitive friend is gone, let me throw in some quick remarks.

6.1 Nondeterministic Sequential Algorithms

One can argue that a real algorithm, being an exact recipe for execution, has to be deterministic, that a nondeterministic algorithm is a contradiction in terms. Even if you buy the first claim (that real algorithms are deterministic), you may want to reject the second (that a nondeterministic algorithm is a contradiction in terms). In reality, nondeterministic algorithms are useful, for example, as higher level descriptions (specifications) of "real" algorithms.

Imagine now that you execute a nondeterministic algorithm A. In a given state, you may have several alternatives for your actions and you have to choose one of the available alternatives. The program of A gives you no instructions how to make the choice. What can you do? Not much. Somebody should make the choice for you. In our setup, the active environment will make the choices. External functions can be used to manifest the choices made by the active environment. In that sense, nondeterminism has been handled already.

It may be convenient though to pretend that the sequential algorithm itself chooses among nondeterministic alternatives. Taking into account that one-step transformation should be bounded work, it is natural to require that nondeterminism is bounded, so that, at each step, the choice is limited to a bounded number of alternatives. One example is nondeterministic Turing machines.

The programming language of nondeterministic sequential ASMs is the programming language of deterministic sequential ASMs enriched with a choose-among constructor that allows one to construct rules of the form

```
choose among
    $R_1$
    ...
    $R_k$
endchoose
```

where $R_1, \ldots, R_k$ are previously constructed rules. It is easy to see that Theorem 1 remains valid in the nondeterministic sequential case.

6.2 Beyond the Sequential Paradigm

I would like to emphasize that general ASMs are not necessarily sequential. They can be parallel, distributed and can involve explicit unbounded nondeterminism [Gurevich 1995]. There are also real-time ASMs. Further, a computer system may involve numerous abstraction levels, in which case it is described not by a single ASM but by a hierarchy of ASMs. The ASM computation model gave rise to a formal method applied in hardware and software engineering [Börger 1995]. All this is beyond the scope of this article. Again, we refer the reader to [Börger and Huggins 1998, ASM Michigan, ASM Paderborn]. Generalizations of the sequential ASM thesis will be addressed elsewhere.

6.3 Higher-Order Structures

First-order structures are versatile and provide the necessary generality. (We use first-order structures without limiting ourselves to first-order logic, which is a common practice in mathematics; think for example about graph theory, group theory or set theory.) In particular, first-order structures support informal typing. However, there are reasons (e.g. debugging) to introduce formal, explicit typing. Many programming languages went this way. Typed ASMs can be found e.g. in [Del Castillo, Gurevich and Stroetmann 1998]. Typing leads naturally to (what logicians call) higher-order structures.

6.4 Positive Information Only

In 1970, E. F. Codd introduced the relational database model [Codd 1970] according to which database states *by definition* are first-order structures, albeit purely relational. A relation is thought of as a set and represented by a table that contains only positive information; a tuple $\bar{a}$ does not belong to a relation R if it differs from every tuple in R. Assuming this positive-information-only representation, one can reasonably argue that the work of adding a new element to a relational structure is bounded. The positive-information-only representation can be extended naturally to structures with functions provided that every function has a default value.

6.5 The Frugal Syntax of ASMs

The sequential ASM thesis speaks about the expressive power of ASMs. By expressive power I mean here the power to model algorithms in step-for-step fashion. One may argue that there are programming languages with parallelism whose expressive power exceeds that of the ASM programming language. In that connection, it may be surprising how frugal the ASM syntax is. In my eyes, the frugal syntax, clear semantics and rich expressivity of the ASM programming language make it an appetizing vehicle for specification, verification, design, etc.

The syntax of sequential ASM rules can be further simplified by means of normal forms. One normal form can be derived from the proof of Theorem 1. Another normal form is a do-in-parallel block of conditional updates. One needs, however, to balance logical simplicity and programming convenience. In this connection see the Pragmatic Occam Razor in [Gurevich 1995].

6.6 Denotational Semantics and ASMs

In the denotational approach, the meaning of a program is an element of some mathematical domain. That works well in many simple cases. For example, the meaning of an ASM program at a given state, in the sense of firing the program just once, is a set of updates. In general, however, the meaning of a program may be way too hard to compute. Consider for example the halting problem for Turing machines that take no input. It is reasonable to assume that the meaning of such a machine includes the indication whether it halts or does not halt, and thus the meaning is not computable. In practice, the question that people address usually in denotational papers is not what is the meaning of a program but what is the type of that meaning, hence the close and fruitful connection between denotational semantics and type theory. The ASM approach is less ambitious but more realistic. A given algorithm is modeled by an ASM (or a hierarchy of ASMs); that gives a precise meaning to the algorithm. But if you want to establish some properties of your algorithm, you need to analyse the mathematical model.

Lowering of ambition has its advantages. In the denotational literature, usually, the meaning of a program reflects only the input-output behavior of the program. Accordingly, the meaning of a nonterminating program is the bottom element of the appropriate mathematical domain. In the ASM approach, the input-output behaviour is only one aspect of a program, and meaningful computations can be nonterminating. Very often, in denotational literature, a particular approach is explained on a toy programming language, and it is not clear how to extend the approach to the languages of real-world programs, say, to the C programming language. The ASM community tends to deal with real-world languages. For example, Egon Börger explained full Prolog in [Börger 1990a, 1990b, 1992]. Using a successive refinement method, Börger and Rosenzweig established the correctness of the standard Prolog implementation [Börger and Rosenzweig 1994]. The C programming language has been addressed in [Gurevich and Huggins 1992]. The extension of the denotational approach to distributed computing is problematic. The extension of the ASM approach to distributed computing was natural [Gurevich 1995].

6.7 Algebraic Specifications and ASMs

In the science of universal algebra, an algebra is a (possibly many-sorted) first-order structure whose vocabulary contains (in addition to the symbols of sorts) only function symbols. An abstract data type is an isomorphism class of algebras, and an algebraic specification is a description of one or more such abstract data types; see [Wirsing 1990]. The descriptions are typically systems of term equations. How does a system E of equations define an algebra? One popular solution is to take the initial algebra in the category of algebras defined by E. The algebra of terms (of the vocabulary of E) factored over E (so that terms t_1, t_2 are made equal if the equality $t_1 = t_2$ follows from E) is such an initial algebra.

The equational approach, so natural in algebra [Tarski 1966], has been used widely in programming theory. In particular it was suggested to describe program states by means of equations. Here one encounters an obvious limitation. While more standard data structures admit simple equational characterizations, an easy information argument shows that most states do not admit simple axiomatizations of any fixed type. Furthermore, one does not need to go far to find natural states that do not suit the equational approach well. Think about finite graphs or databases for example. The equational approach is too restrictive to deal with the states of arbitrary sequential (let alone distributed) algorithms.

In the ASM approach, relations are viewed as special functions, and thus the states of ASMs can be viewed as algebras as well. (Accordingly a run of an ASM can be seen as an evolution of an algebra, hence the older term "evolving algebra" for an abstract state machine.) However, in the ASM approach, there is no prescribed way to represent a state. Again, the ASM approach is less ambitious and more realistic. In cases when we need state representations, e.g. to build ASM tools, we use representations most convenient for the purpose. Term equations may be a part of the picture, e.g. as integrity constraints.

6.8 Some Related or Unrelated Work

I do not know of any other author who has worked explicitly on the task of capturing sequential algorithms in the strong sense discussed above. Turing's thesis was elaborated by Robin Gandy in [Gandy 1980]. There is a similarity between Unity [Chandy and Misra 1988], which builds on Dijkstra's guarded command approach [Dijkstra 1976], and parallel ASMs. Since parallelism is so inherent in Unity and since this article is devoted primarily to sequential ASMs, I will delay the ASM vs. Unity issue to another occasion.

The concepts involved in the definition of sequential ASMs are few. With the notable exception of the do-in-parallel constructor, they are very common. Consequently, one may not see much novelty in the notion of ASM. Lately several colleagues, including Cremers, Ganzinger and Rajlich, brought to my attention their articles, in particular [Cremers and Hibbard 1978, Cremers and Hibbard 1985, Ganzinger 1983, Rajlich 1977], supposedly similar to the ASM approach. I certainly want to be fair and give my colleagues their due. However, I have not found any of these articles particularly similar to the ASM approach. For brevity, let me address only one of the similarity claims.

[Ganzinger 1983] is a denotational semantics article. Its example programming language is a while-language with procedures and abstract data types. The meaning of a while statement is the least fixed point of a recursive equation. If the loop does not terminate, its meaning is the bottom element of the appropriate category of algebras. [Ganzinger 1983] is also an algebraic specification article. Its declared goal is to extend denotational semantics to "programs over abstractly specified data types". The program states in [Ganzinger 1983] are initial algebras in categories defined by systems of equations. Since ASM states are algebras as well, there is a similarity there. However, as I explained above, in the remark on Algebraic Specifications vs. ASMs, the similarity hides a substantial dissimilarity. In the spirit of algebraic specifications, Ganzinger proves that the state transformations described by his programs are functorial. But the functoriality comes at a price: a severe restriction on equality tests in the programming language. Is the price worth paying? What does functoriality buy us? There is no explanation in the paper.

6.9 The Early ASM Paper Trail

In 1982, I moved from logic to computer science (and to Michigan). Teaching Pascal and noting that different compilers interpret Pascal differently, I wondered — as many did before me — what are programming languages, mathematically speaking. For a while, I studied the wisdom of the time, especially denotational semantics and algebraic specifications. Certain aspects of those approaches appealed to this logician and former algebraist. I felt though that neither approach was realistic. It occurred to me that, in a sense, Turing solved the problem of the semantics of programs. Each program can be simulated and thus given a precise meaning by a Turing machine. The obvious problem was that Turing machines were so low level. I worried also about reflecting properly the bounded resources of a computation. All that led me to "Reconsidering Turing's Thesis: Toward More Realistic Semantics of Programs" [Gurevich 1984] and then to "A New Thesis" [Gurevich 1985]. "Logic and the challenge of computer science" [Gurevich 1988a] was written about the same time though the publication of the book was much delayed by the publisher. Another early paper was "Algorithms in the world of bounded resources" [Gurevich 1988b]. The first full-language study resulted in the doctoral dissertation "Algebraic operational semantics and Modula-2" [Morris

1988] with an extended abstract [Gurevich and Morris 1987]. During the 1989 conference on Computer Science Logics (CSL'89), Egon Börger presented the first part of his fundamental study of full Prolog by means of ASMs [Börger 1990a], and Gurevich and Moss presented the first distributed ASM [Gurevich and Moss 1990].

My early ASM work contained no references to [Kolmogorov and Uspenski 1958] or [Schönhage 1980] because I learned about these two articles later. Neither article puts forward a foundational thesis explicitly. The speculation in [Gurevich 1988c] about the thesis behind the notion of Kolmogorov-Uspenski machine was later confirmed by Vladimir Uspenski [Uspenski 1992, page 396].

Acknowledgements

I owe much to the ASM community and especially to Egon Börger, Andreas Blass and Dean Rosenzweig. Egon Börger was the first colleague to recognize the potential of abstract state machines. He contributed greatly to the ASM theory and practice; much of ASM application work was done under his leadership. All aspects of this paper were thoroughly discussed with Andreas Blass who provided me with a sanity check; to a great extent, my foundational work on ASMs became joint work with Andreas. Dean Rosenzweig never failed to provide imaginative sharp criticism. This particular article benefitted from remarks by Jim Huggins, Peter Päppinghaus and Chuck Wallace.

References

ASM Michigan http://www.eecs.umich.edu/gasm, the ASM Michigan website maintained by James K. Huggins.

ASM Paderborn http://www.uni-paderborn.de/cs/asm.html, the ASM Paderborn website maintained by Uwe Glässer.

Börger 1990a Egon Börger, "A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control", in E. Börger et al. (editors), CSL'89 (3rd Workshop on Computer Science Logic), Lecture Notes in Computer Science, volume 440, pages 36–64, Springer, 1990.

Börger 1990b Egon Börger, "A Logical Operational Semantics of Full Prolog. Part II: Built-in Predicates for Database Manipulation", in B. Rovan (editor), MFCS'90 (Mathematical Foundations of Computer Science), Lecture Notes in Computer Science, volume 452, pages 1–14, Springer, 1990.

Börger 1992 Egon Börger, "A Logical Operational Semantics of Full Prolog. Part III: Built-in Predicates for Files, Terms, Arithmetic and Input-Output", in Y. Moschovakis (editor), "Logic From Computer Science", Berkeley Mathematical Sciences Research Institute Publications, volume 21, 17–50. Springer, 1992.

Börger 1995 Egon Börger, "Why Use Evolving Algebras for Hardware and Software Engineering?". in M. Bartosek et al. (editors), SOFSEM'95 (22nd Seminar on Current Trends in Theory and Practice of Informatics), Lecture Notes in Computer Science, volume 1012, pages 236–271. Springer, 1995.

Börger and Huggins 1998 Egon Börger and James K. Huggins, "Abstract State Machines 1988–1998: Commented ASM Bibliography", in H. Ehrig (editor), Formal Specification Column, Bulletin of European Assoc. for Theor. Computer Science, Number 64, February 1998, 105–128. For an updated version see [ASM Michigan].

Börger and Rosenzweig 1994 Egon Börger and Dean Rosenzweig, "The WAM — Definition and Compiler Correctness", pp 20–20, in C. Beierle and L. Plümer (editors), "Logic Programming: Formal Methods and Practical Applications", Series: Studies in Computer Science and Artificial Intelligence, North-Holland, 1994.

Chandy and Misra 1988 K. Mani Chandy and Jayadev Misra, "Parallel Program Design", Addison-Wesley, 1988.

Church 1936 Alonzo Church, "An unsolvable problem of elementary number theory", American Journal of Mathematics 58 (1936), 345–363.

Codd 1970 E. F. Codd, "A Relational Model for Large Shared Data Banks", Communications of ASM 13:6, 377–387.

Cook and Reckhow 1973 S. A. Cook and R. A. Reckhow, "Time-Bounded Random Access Machines", J. Comp. Systems Sci. 7 (1973), 354–475.

Cremers and Hibbard 1978 Armin B. Cremers and T. N. Hibbard, "Formal Modeling of Virtual Machines", IEEE Transactions of Software Engineering 4, 426–436, 1978.

Cremers and Hibbard 1985 Armin B. Cremers and T. N. Hibbard, "Executable Specification of Concurrent Algorithms in Terms of Applicative Data Space Notation", in S. Y. Kung et al. (editors), "VLSI and Modern Signal Processing", Chapter 12, 200–223, Prentice-Hall, 1985.

Del Castillo, Gurevich and Stroetmann 1998 Giuseppe Del Castillo, Yuri Gurevich and Karl Stroetmann, "Typed Abstract State Machines", [ASM Michigan].

Dijkstra 1976 E. W. Dijkstra, “A Discipline of Programming”, Prentice Hall, 1976.

Gandy 1980 Robin Gandy, “Church’s Thesis and Principles for Mechanisms”, in J. Barwise et al. (editors), ”The Kleene Symposium”, North-Holland, 1980, 123–148.

Ganzinger 1983 Harald Ganzinger, “Denotational Semantics for Languages with Modules”, in D. Bjørner (editor), Formal Descriptions of Programming Concepts — II, North-Holland, 1983, 3–20.

Gurevich 1984 Yuri Gurevich, “Reconsidering Turing’s Thesis: Toward More Realistic Semantics of Programs”, Technical report CRL-TR-36-84, EECS Department, University of Michigan.

Gurevich 1985 Yuri Gurevich, “A New Thesis”, Abstracts, American Mathematical Society, August 1985, page 317.

Gurevich 1988a Yuri Gurevich, “Logic and the challenge of computer science”, in E. Börger (editor), “Current Trends in Theoretical Computer Science”, Computer Science Press, 1988, 1–57.

Gurevich 1988b Yuri Gurevich, “Algorithms in the world of bounded resources”, in R. Herken (editor), “The universal Turing machine - a half-century story” Oxford University Press, 1988, 407–416.

Gurevich 1988c Yuri Gurevich, “Kolmogorov machines and related issues”, Bulletin of European Assoc. for Theor. Computer Science, Number 35, June 1988, 71–82. Reprinted in G. Rozenberg and A. Salomaa (editors), “Current Trends in Theoretical Computer Science”, World Scientific, 1993, 225–234.

Gurevich 1991 Yuri Gurevich, “Evolving Algebras: An Attempt to Discover Semantics ”, Tutorial, Bull. EATC 43 (1991), 264–284. Reprinted in G. Rozenberg and A. Salomaa (editors), “Current Trends in Theoretical Computer Science”, World Scientific, 1993, 266–292.

Gurevich 1995 Yuri Gurevich, “Evolving Algebra 1993: Lipari Guide”, in “Specification and Validation Methods”, Ed. E. Börger, Oxford University Press, 1995, 9–36. See also “May 1997 Draft of the ASM Guide” at [ASM Michigan].

Gurevich and Huggins 1992 Y. Gurevich and J. Huggins, “The Semantics of the C Programming Language”, CSL’92 (Computer Science Logics), Springer Lecture Notes in Computer Science 702, 1993, 274–308.

Gurevich and Morris 1987 Yuri Gurevich and James Morris, “Algebraic operational semantics and Modula-2”, CSL’87 (1st Workshop on Computer

Science Logic), Springer Lecture Notes in Computer Science 329 (1988), 81–101.

Gurevich and Moss 1990 Yuri Gurevich and Larry A. Moss, “Algebraic Operational Semantics and Occam”, CSL’89, 3rd Workshop on Computer Science Logic, Springer Lecture Notes in Computer Science, no. 440 (1990), 176–192.

Gurevich and Spielmann 1997 Yuri Gurevich and Marc Spielmann, “Recursive Abstract State Machines”, Springer Journal of Universal Computer Science (JUCS), Vol. 3, No. 4, April 28, 1997, 233–246.

Kolmogorov and Uspenski 1958 A. N. Kolmogorov and V. A. Uspenski, “On the definition of algorithm”, Uspekhi Mat. Nauk 13 (1958), 3–28 (Russian). AMS Transl. 29 (1963), 217–245.

Rajlich 1977 Vaclav Rajlich, “Theory of Data Structures by Relational and Graph Grammars”, in Automata, Languages, and Programming (A. Salomaa and M. Steinby, editors), Lecture Notes in Computer Science 52, Springer Verlag, 1977, 391–411.

Savage 1998 John E. Savage, “Models of Computation: Exploring the Power of Computing”, Addison Wesley Longman 1998.

Schönhage 1980 Arnold Schönhage, “Storage Modification Machines”, SIAM J. on Computing 9 (1980), 490–508.

Tarski 1966 Alfred Tarski, ”Equational logic and equational theories of algebras”, pp. 275–288, in H. A. Schmidt et al (editors), “Contributions to Mathematical Logic: Proceedings of the Logic Colloquium, August 1966, Hannover”, Series: Studies in Logic and the Foundations of Mathematics, North-Holland Publishing Co., Amsterdam, 1968.

Turing 1936 Alan Turing, “On computable numbers, with an application to the Entscheidungsproblem”, Proc. of London Mathematical Society 2, no. 42 (1936), 230–236, and no. 43 (1936), 544–546.

Uspenski 1992 Vladimir A. Uspenski, “Kolmogorov and Mathematical Logic”, J. Symbolic Logic 57:2, June 1992, 385–412.

Wirsing 1990 Martin Wirsing, “Algebraic Specifications”, in J. van Leeuwen (editor), Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, Elsevier and MIT Press, 1990, 675–788.