THE LOGIC IN COMPUTER SCIENCE COLUMN[*]
by
Yuri GUREVICH

Algorithms vs. Machines

Andreas Blass[†] Yuri Gurevich[‡]

Abstract

Yiannis Moschovakis argues that some algorithms, and in particular the mergesort algorithm, cannot be adequately described in terms of machines acting on states. We show how to describe the mergesort algorithm, on its natural level of abstraction, in terms of distributed abstract state machines.

1 Prelude

Quisani: I have a question about the ASM thesis, the claim that every algorithm can be expressed, on its natural level of abstraction, by an abstract state machine (ASM). Do you still believe this thesis?

Authors: Yes. In fact, there has been some recent progress [2], extending the proof from the sequential algorithms covered in [4] to parallel algorithms.

[*]Bulletin of Euro. Assoc. for Theor. Computer Sci., Number 77, June 2002, 96–118

[†]Partially supported by NSF grant DMS–0070723 and by a grant from Microsoft Research. Address: Mathematics Department, University of Michigan, Ann Arbor, MI 48109–1109, U.S.A., ablass@umich.edu. This paper was written during a visit to Microsoft Research.

[‡]Microsoft Research, One Microsoft Way, Redmond, WA 98052, U.S.A., gurevich@microsoft.com

To be precise, we deal with parallel algorithms that operate in sequential time and have bounded sequentiality within each step.

Q: Wait a minute; I'm confused. You go beyond the sequential case yet still assume sequential time?

A: Right. "Sequential time" means only that the algorithm proceeds in a sequence of discrete steps. "Sequential" means in addition that only a bounded amount of work is done in each step. So a sequential algorithm obeys all the postulates in [4], not just the sequential time postulate. (We think "small-step algorithm" may be a better name than "sequential algorithm," but the latter terminology is prevalent in the literature.)

Q: So the parallel algorithms in [2] have discrete steps but can do an unbounded amount of work in each step.

A: Right, provided the work is done in parallel; we don't allow (except in one section of the paper) unbounded sequentiality within a step.

Q: I came across a provocative article "What is an algorithm?" by Yiannis Moschovakis [8], casting doubt on the ASM thesis. The article begins, "When algorithms are defined rigorously in Computer Science literature (which only happens rarely), they are generally identified with *abstract machines.*" Then Moschovakis argues that "this does not square with our intuitions about algorithms," and he speaks about the problem of defining algorithms correctly. "This problem of *defining algorithms* is mathematically challenging, as it appears that our intuitive notion is quite intricate and its correct, mathematical modeling may be quite abstract—much as a 'measurable function on a probability space' is far removed from the naive (but complex) conception of a 'random variable'. In addition, a rigorous notion of algorithm would free many results in complexity theory from their dependence on specific (machine) models of computation, and it might simplify their proofs." He proposes "a plausible solution" according to which "algorithms are *recursive definitions* while machines model [are] *implementations,* a special kind of algorithms."

A: It's not clear *a priori* that the problem of rigorous definition of *algorithm* is solvable; does this intricate intuitive notion lend itself to mathematical definition? Is there one mathematical model for all algorithms? Like Moschovakis, we hope that the answer is positive; in fact this was one motivation for the ASM project. From the beginning, this project was oriented toward real-world computer systems. Nowadays, our group at Microsoft Research works on practical (and executable!) specification of large, complicated, highly interactive, and highly distributed systems related to Microsoft's .NET initiative. In this context, an approach via recursive equations looks quite impractical, but an approach via machine models is working well.

Q: Moschovakis seems to be interested more in a general definition of algorithms for theoretical purposes (like complexity theory) than in practical applicability. Though he doesn't say so in his paper, he would probably think that your practical work forces you to think of implementations more than of algorithms. He is quite willing to identify implementations with machines, but he says that algorithms are something more general.

What it boils down to is that you say that an algorithm is a machine and Moschovakis says it is not.

A: A reasonable conclusion would be that different meanings are being attached to "algorithm" or "machine" or both.

Q: Or, as President Clinton suggested, to "is."

A: The semantics of "is" is in general not so simple but there seems to be no disagreement about it here; let's stick to "algorithm" and "machine." In fact, it seems that Moschovakis uses both of these words differently than we do.

2 Sequential Machines

Q: In the case of "machine," it should be easy to figure out what's going on. Moschovakis explicitly defines what he calls "an *abstract* (or *sequential*) *machine*" with inputs in a set X and outputs in a set Y. It consists of

- a set S of states,
- an initial state $s_0 \in S$,
- a transition function $\sigma : X \times S \to S$,
- a set $T \subseteq S$ of terminal states, and
- an output function $o : X \times T \to Y$.

Apart from the presence of terminal states and an output function, this looks very similar to the sequential time postulate in [4]. And as for the output, Moschovakis adds in a footnote that "Gurevich, in effect, 'identifies' the output with the computation [the sequence of states] $\ldots$, so he can model algorithms which 'run forever'."

A: That footnote seems to be a misunderstanding. The sequential time postulate of [4] asserts that every sequential algorithm has states, initial states and a transition function. The postulate says nothing about the output. But in [4], states are not just points in the state space as in the case of abstract machines. They have internal structure which may include an output. See for example the ASMs used in [1]. In [2] we define a more general notion of "sequential ASM with output" that allows output not only at the end of the computation but at any step. In any case, the output is not identified with the state or the sequence of states.

Q: I haven't seen [2] yet, but I remember [1]. The sort of ASMs used there, with "output" and "halt" as dynamic functions, seem to fall into the category of the machines described by Moschovakis. In fact I see only two differences between his machines and sequential time algorithms, like sequential time ASMs. Both differences relate to the way inputs are handled, and both are apparently inessential.

First, ASMs have many possible initial states, one for each possible input, while Moschovakis's machines have just one initial state s_0. But Moschovakis's transition function σ is defined on pairs $\langle \text{input}, \text{state} \rangle$, so it can take $\langle \text{input}, s_0 \rangle$ to the state that, from the ASM viewpoint, would be the initial state for that input.

Second, Moschovakis's transition function acts, as I just said, on pairs $\langle \text{input}, \text{state} \rangle$, not just on states. But this difference can be eliminated by redefining "state" to include the input. In fact, if the transition function really uses the input, then you'd insist that the input be remembered as part of the state; I've heard you say often that the state must include everything relevant to the future progress of the computation.

A: Right. There seems to be no essential difference between Moschovakis's description of abstract machines and the sequential time postulate, if the latter is adjusted to demand (rather than merely permit) terminal states and output.

Q: In fact, an earlier paper [7] of Moschovakis, referred to in [8], gives a description of "iterators" that is even closer to the sequential time postulate.

So you and Moschovakis agree about what a *machine* is, and we should look instead at the concepts of *algorithm*.

A: Not so fast! Moschovakis's notion of abstract machine (or iterator) is very similar to the sequential time postulate, but not all machines satisfy that postulate. Specifically, distributed ASMs [5] and other distributed algorithms may not run in sequential time. In computations of such algorithms, one has an initial state, but what happens next may depend on which one of many agents acts first. So there will not in general be a uniquely determined second state. The whole idea of a computation as a single, linear sequence of states is tied to the notion of sequential time.

So it is not surprising that Moschovakis finds abstract machines inadequate to model general algorithms. It could mean simply that not all algorithms are sequential time algorithms.

Q: It would be nice if this fully explained why Moschovakis regards machines as inadequate.

3 Mergesort Equations

Q: Moschovakis uses the example of the mergesort algorithm to show how interesting results can be established just on the basis of a recursive definition without descending to the level of an implementation. He shows that mergesort needs only $O(n \log n)$ comparisons to sort a string of length n. Since he emphasizes this specific example let's look at it and try to understand what's going on.

A: OK. How does Moschovakis define mergesort?

Q: Not surprisingly, he defines it by recursion. It is the function *sort* defined, along with another function *merge*, by the equations:

$$\operatorname{sort}(u) = \begin{cases} u, & \text{if } |u| \le 1, \\ \operatorname{merge}(\operatorname{sort}(h_1(u)), \operatorname{sort}(h_2(u))), & \text{otherwise,} \end{cases} \tag{1}$$

$$\operatorname{merge}(v, w) = \begin{cases} w, & \text{if } v = \varnothing, \\ v, & \text{else, if } w = \varnothing, \\ \langle v_0 \rangle * \operatorname{merge}(\operatorname{tail}(v), w), & \text{else, if } v_0 \le w_0, \\ \langle w_0 \rangle * \operatorname{merge}(v, \operatorname{tail}(w)), & \text{otherwise.} \end{cases} \tag{2}$$

Here u, v, w range over strings on a fixed alphabet, $|u|$ is the length of u, $\le$ is a linear ordering of the alphabet, $h_1(u)$ and $h_2(u)$ are the first and second halves of the string u "appropriately adjusted when $|u|$ is odd", $\varnothing$ is the empty string, $\operatorname{tail}(v)$ is the string obtained from v by deleting its first letter v_0, and $*$ is the concatenation operation on strings.

An abstract machine (in Moschovakis's sense) may implement mergesort but any such implementation will be too specific. It may sort $h_1(u)$ first, or sort $h_2(u)$ first, or interleave the two computations. Whatever it does, it is too specific. In your favorite phrase, there is no abstract machine that implements the mergesort algorithm on its natural abstraction level. I wonder if there is any state-based way to capture the mergesort algorithm on its natural abstraction level.

A: Wait a minute. Equations (1) and (2) give a specification, not an algorithm. As you said, they define the functions sort and merge. But unless some further information is given or assumed, we could "implement" this definition of sort by using quicksort and checking that the result satisfies (1).

Q: You're right. I presume that the intention, in calling these equations an algorithm, is that, given a string u of length greater than 1, you should sort it by splitting it into the two halves $h_1(u)$ and $h_2(u)$, sorting these, and merging the results.

A: With that presumption and an analogous one for computing merge, you have described an algorithm. It can be modeled faithfully by a recursive ASM in the sense of [6]. Recursive ASMs are state-transition systems but need not satisfy the sequential time postulate.

If we model equation (1) by a recursive ASM, temporarily regarding merge as a given function, we get a distributed algorithm that sorts strings in the way you described, namely to sort both halves and then merge the results. It allows runs in which $h_1(u)$ is sorted first, other runs in which $h_2(u)$ is sorted first, and others in which these two operations are performed in parallel, or by interleaving subcomputations for the substrings $h_i(h_j(u))$ and for smaller substrings.

If we don't regard merge as a given function but include its computation in our recursive ASM, then more implementations become available, for example interleaving parts of one sorting operation with parts of another merging operation.

Q: Runs like these correspond to the various implementation options mentioned by Moschovakis in [8]. So I'll be asking you to explain recursive ASMs and their runs.

4 Implementations of Recursive Equations

Q: But first, I wonder whether we (and Moschovakis) shouldn't consider a broader class of implementations.

A: What sorts of implementations do you have in mind?

Q: Well, these recursive equations are to be interpreted as defining the least fixed-point of an operator on partial functions, as explained in [8, Sections 2 and 4]. And the natural way to compute least fixed-points is the iterative process described, for example, in [7, Section 6.7]. In the mergesort example, this amounts to first sorting all strings of length 1, then sorting all strings of length 2, then those of lengths 3 and 4, and so on until you've sorted the given string u. At each step, sort strings by applying equation (1) to previously sorted shorter strings. (I'm using merge as a given function, but a similar description would apply if it were computed recursively along with sort.)

In the case of mergesort, this bottom-up process is inefficient, because it sorts a lot of strings not relevant to the given instance u. But for other algorithms, it may make good sense, and it may be more efficient than the top-down process. For example, to compute the Fibonacci numbers, defined recursively by

$$F(n) = \begin{cases} 1, & \text{if } n = 1 \text{ or } n = 2 \\ F(n-1) + F(n-2), & \text{otherwise,} \end{cases}$$

the bottom-up approach works well but a (naive) top-down approach recomputes the same values $F(k)$ absurdly often.

But apart from questions of efficiency, it seems reasonable to regard the bottom-up process as an implementation. How does this look from the point of view of recursive ASMs?

A: One can use nondeterministic recursive ASMs to include more of mergesort's implementations, but there is no reason to believe that every declarative specification can be faithfully captured by an algorithm. Fairness, for example, is a legitimate ingredient for specifications but is not algorithmic.

As for the bottom-up implementation of mergesort, didn't you say that Moschovakis proves that mergesort needs only $O(n \log n)$ comparisons to sort a string of length n? The bottom-up implementation you described would use many more comparisons, since it sorts a lot of irrelevant strings.

Q: We'd better look more closely at Moschovakis's definition of "needs only q comparisons." He gives a precise mathematical definition of this concept, along the following lines. Take the recursive equations (1) and (2) and replace every mention of the order relation $x \leq y$ by $c(x, y)$ where c is a parameter representing a partial binary relation on the alphabet. This means that $c(x, y)$ can be true or false or undefined. We're interested in c's that are subrelations of $\leq$ in the sense that if $c(x, y)$ is true (resp. false) then $x \leq y$ (resp. $x \not\leq y$), i.e., c is obtained from $\leq$ by making it undefined at certain arguments. Since c is only a part of $\leq$ (and since the recursion equations define an operator monotone[1] in c), the functions sort_c and merge_c defined using c are subfunctions of sort and merge. Now what Moschovakis proves is that, for every string u of length n there exists a subrelation c of $\leq$ such that $\mathrm{sort}_c(u)$ is defined (and thus equal to $\mathrm{sort}(u)$) and such that c is defined on only $O(n \log n)$ pairs. Intuitively, this means that the value of $\mathrm{sort}(u)$ is computed, via the recursive equations, using only those comparisons that are given by c, and thus using at most $O(n \log n)$ comparisons.

A: This looks like a perfectly reasonable definition of information usage by recursive computations. We especially like it because it seems to match what

[1]To be precise, the clause "if $v_0 \leq w_0$", the only occurrence of $\leq$ in (1) and (2), becomes "if $c(v_0, w_0)$" and is to be understood as follows. If $c(v_0, w_0)$ is true then this case applies; if $c(v_0, w_0)$ is false then move on to the next case; and if $c(v_0, w_0)$ is undefined then the result of this merge_c is undefined. Monotonicity means that if we change any c to be defined at more arguments, without changing the true or false values it already has, then sort_c and merge_c only become more defined if they change at all.

we'd get using recursive ASMs.

It means, however, that either this sort of resource bound for an algorithm might fail to carry over to its implementations or else your bottom-up implementation of mergesort isn't really an implementation. For the bound on the number of comparisons is, as we saw, violated by the bottom-up implementation.

Q: Moschovakis intends that the comparison count, once established for the general mergesort algorithm, applies automatically to all its implementations. Specifically, for algorithms like mergesort[2] a mathematical definition of "implementation" is given in [7]. In [8], Moschovakis briefly describes this notion of "implementation" and says that "it behaves well with respect to resource use, e.g., it justifies extending to all implementations of the mergesort the comparison counts established for the algorithm."

A: So it seems that the bottom-up "implementation" that you suggested would not qualify as an implementation according to the definition in [7]. Maybe we should compare it with the definition to see just what goes wrong[3].

Q: That may get a bit complicated, but I tried to improve my understanding of the definition of "implementation" by checking it in the case of the factorial function on the natural numbers. I used the obvious recursive definition

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n \cdot (n-1)!, & \text{otherwise.} \end{cases}$$

The first implementation I tried was a top-down computation given by the program

```
step
   read $n$
   var $p = 1$         // introduce a variable $p$, initially 1
   var $x = n$         // introduce a variable $x$, initially $n$
step until $x = 0$
   $p := p \cdot x$
   $x := x - 1$
step
   Result $:= p$.
```

[2] "Like mergesort" here means that the range of outputs of the algorithm is an ordinary set, as opposed to a general complete poset, which is allowed in [8].

[3] The material from here to the end of this section can be safely skipped.

A: Both the recursive equation and the program look OK. Does the program implement the recursion by Moschovakis's definition of "implement"?

Q: Yes. With a couple of pages of work, I was able to verify the definition.

Next, I tried a bottom-up approach. Rather than write out a program, I'll describe this as an abstract machine in Moschovakis's sense. The states are the initial state s_0 and all pairs $\langle n, s\rangle$ where $n \in \mathbb{N}$ and s is a partial function from $\mathbb{N}$ to itself. The transition function for input m sends s_0 to $\langle m, \varnothing\rangle$ and sends $\langle n, s\rangle$ to $\langle n, s'\rangle$ where

$$s'(k) = \begin{cases} 1, & \text{if } k = 0, \\ k \cdot s(k-1), & \text{otherwise.} \end{cases}$$

Terminal states are the pairs $\langle n, s\rangle$ with n in the domain of s, and the output function sends $\langle m, \langle n, s\rangle\rangle$ (where m is the input and $\langle n, s\rangle$ a terminal state) to $s(n)$.

A: We agree that this computes the factorial function in a bottom-up manner. The successive states of the computation have, in their second components, longer and longer initial segments of the factorial function. But the machine looks awkward. The first component of a state $\langle n, s\rangle$ just keeps track of the input, which plays no role during the main computation. And the output is given by a function of input and state, so you don't need to remember the input as part of the state in order to produce the output.

Q: True, but in Moschovakis's definition the input is not available when you determine whether a state is terminal. The definition of "terminal" in my machine is the only place where I need that the state remembers the input.

A: We don't think Moschovakis would mind an amendment to his definition, making the input available when deciding whether a state is terminal.

Q: I agree, but I was trying to understand exactly his definitions. So I refrained from amending anything.

I worked out what Moschovakis's definition would require in order to call this abstract machine an implementation of the factorial. As far as I can see, the requirement is not satisfied.

A: You say "as far as I can see." Does that mean there's no easy way to decide whether a machine implements an algorithm?

Q: No, the definition of "implements" requires the existence of two functions. One maps inputs to inputs and seems to be just a matter of reformatting the input. In the present situation, where both the algorithm and the machine take as input the number whose factorial is to be computed, this function would be the identity. The other function, though, is more complicated. Its domain consists, in our example, of partial functions from $\mathbb{N}$ to $\mathbb{N}$, and its range consists of partial functions from states to numbers, subject to some requirements.

A: That sounds pretty complicated. Maybe we should look closely at the definition and try to understand it better.

Q: We probably should, but it might be more efficient if I first try to understand examples better and then discuss it with you. Besides, time is limited and I still want to hear about recursive ASMs and their runs.

A: OK. Anyway, it makes sense that a bottom-up computation of the factorial doesn't count as an implementation for Moschovakis. We already saw that a bottom-up computation of mergesort had better not count as an implementation because it violates the resource bound.

5 A Recursive ASM for Mergesort

A: Now about recursive ASMs, let's begin with the simplest case, which is treated in [6, Section 2], and which suffices to handle mergesort.

The key idea is that recursive computations are to be viewed as a special case of distributed computation.

Q: So in the case of mergesort, there would be one agent for the function sort and another for merge.

A: No, there would in general be many agents for each function. The idea is that to call a function means to create a new "vassal" agent whose job is to execute that function call. The agent who calls the function — we call it the "lord" of the vassal[4] — passes the appropriate arguments for the function to the vassal, and when the vassal's work is complete it passes the computed value of the function back to the lord.

Notice that the information often stored in a stack, telling who reports to whom, is here stored locally. Each vassal just remembers who its lord is.

[4]In [6] the terminology was "master" and "slave" but recently we have used "lord" and "vassal." We also changed the `Return` of [6] to `Result`, since a noun is more appropriate.

Q: Why don't you use a stack? Standard implementations of recursion do.

A: Stack implementations work as long as recursive calls occur sequentially, one at a time. But sometimes it is useful and natural to make many recursive calls in parallel. Our stack-free approach makes this work well.

In [6], a convenient notation was developed for recursive ASMs. Rather than repeating details here, we'll just exhibit the recursive ASM that expresses the mergesort algorithm (see Figure 1). Most of the program will be self-explanatory, and we'll be happy to answer your questions about the rest. (To avoid clutter, we omitted lots of end-markers like `endif` that the notation of [6] requires. Just assume that the beginning of every line closes all open blocks that started with the same or greater indentation, except that `else` and `elseif` don't close the open `if`-block that began with the same indentation. Equivalently, just use common sense.)

Q: Let me make sure I'm reading this program correctly. The two blocks labeled `rec` tell how to compute the functions sort and merge, so they are the programs for two sorts of vassals. The lines preceding the first `rec` are the main program, which just starts the recursive computation and puts the final result into `Output`. Whenever any agent executes a rule involving `Sort`, a vassal is created to carry out the sorting, and similarly for `Merge`. The dynamic, nullary symbol `Result` in a vassal's program gets the value to be passed to the lord; the lord uses this result as the value of the term for which he created the vassal.

A: Right. We also used the convention that each agent starts with `Mode` equal to `Initial` and finishes when `Mode` becomes equal to `Final`.

Q: But something's wrong with dynamic symbols like `Mode` and `Result` that are shared by all the agents. Updates by one agent will interfere with other agents' computations.

A: Those locations are not really shared. Each agent has his own copy of these memory locations. In the official notation of [6], there is a nullary `Mode` used by the main program (the first two lines in our example) and a unary function, say `Mode′(Me)` such that each agent a uses `Mode′`(a) where we have written simply `Mode` in the program. But the primes[5] and the `Me`'s required by this official notation are so cumbersome that a convention was

[5]We could avoid the primes by having an initial agent that executes the main program; then `Mode` would be a unary function everywhere.

Figure 1: Mergesort

```
if Mode=Initial then
   Output := Sort(Input), Mode := Final

rec Sort($u$)
   if Mode=Initial then
      if $|u| \leq 1$ then
         Result := $u$, Mode := Final
      else
         $v$ := Sort($h_1(u)$)
         $w$ := Sort($h_2(u)$)
         Mode := Halfway
   elseif Mode=Halfway then
      Result := Merge($v, w$), Mode := Final

rec Merge($v, w$)
   if Mode=Initial then
      if $v = \varnothing$ then Result := $w$
      elseif $w = \varnothing$ then Result := $v$
      elseif $v_0 \leq w_0$ then Result := $\langle v_0 \rangle *$ Merge(tail($v$), $w$)
      else Result := $\langle w_0 \rangle *$ Merge($v$, tail($w$))
   Mode := Final
```

adopted in [6] whereby they can be omitted. The general convention is that any dynamic function occurring in the program of a vassal is understood as implicitly having an extra argument `Me` (and a prime if needed to distinguish it from a symbol in the main program).

Q: With that convention, the program looks good to me. Let me check how the information flow works when we sort a string u. For simplicity, I'll assume (as Moschovakis does in [7, 8] when he checks details) that the length of u is a power of 2, say $n = 2^m$. The main program creates a sort-agent, whose result will be the final output, and it passes the input u to this agent. This agent, executing the program in the block `rec Sort`(u) and being in `Initial` mode, creates two vassals, each sorting half of u, i.e., a string of length 2^{m-1}. Then these create vassals who sort strings of length 2^{m-2}, and so on. If I cleverly start numbering levels of this tree of vassals at 0, then there are 2^k sort-agents at level k, each sorting a string of length 2^{m-k}. That continues until at level m there are 2^m agents sorting strings of length 1. They don't need further vassals, as a string of length 1 is already sorted. So these agents pass their (sorted) strings to their (immediate) lords. Each of these lords can now assign values to its v and w and, having completed the "`Mode` = `Initial`" part of its program, goes into `Mode` = `Halfway`. It therefore creates a merge-vassal. In general, each sort-agent in the tree (except for leaves) will get a sorted string from each of its two vassals, assign values to v and w, go into `Mode` = `Halfway`, and create a merge-vassal to merge those two strings.

Merge-agents form chains rather than trees, since each one (unless it is at the bottom of its chain) extracts one element from one of its two sequences and passes the rest to a single vassal to merge. When it gets the vassal's result, it prepends the extracted element and passes the result to its lord. At the bottom of each chain is a merge-agent one of whose two input sequences is empty, and this agent just returns the other input sequence to its lord.

A: Right. Now it's fairly easy to estimate the number of comparisons made during this computation.

Each merge-agent with nonempty input sequences makes exactly one comparison, to decide which sequence's head is earlier, and a sort-agent makes no comparisons. So the problem is to estimate the number of merge-agents with nonempty inputs. A sort-agent at level $k < m$ creates a merge-agent whose job is to merge two sequences of length 2^{m-k-1}. Each time a merge-agent creates a vassal, the total length of the sequences to be merged by the vassal is 1 less than the total length for the lord. So the chain of merge-agents descending from a sort-agent at level k consists of at most 2^{m-k} agents, all but the last of which make a comparison. So from one sort-agent at level k we get fewer than 2^{m-k} comparisons; from all 2^k sort-agents at level k, we get fewer than 2^m comparisons; and so altogether, since there are m levels to consider, there are fewer than $2^m \cdot m$ comparisons. Since $|u| = n = 2^m$, this bound is $n \cdot \log n$.

Q: So we get the same estimate as Moschovakis. In fact, the “big O” in his estimate is unnecessary.

A: That's because of the simplifying assumption that n is a power of 2; for general n, there will be a small constant factor in the estimate, and so the “big O” is needed after all.

Q: Of course. I forgot that Moschovakis also gets $n \cdot \log n$ without a big O when n is a power of 2.

You claimed that the recursive ASM in Figure 1 faithfully models the algorithm described by Equations (1) and (2)

A: Don't forget that the equations alone just define the functions. To describe an algorithm, they must be supplemented with an understanding that the sort function is to be computed by splitting the input, sorting the halves, and merging the results — as the equations suggest but do not demand.

Q: Right. The agreement between the estimate of comparisons that I made here and Moschovakis's bound lends some support to your claim. In fact, I see some additional support in the proofs of these estimates. Admittedly, your computation doesn't look exactly like the one Moschovakis gives, but they match up pretty closely. Moschovakis's proof works with statements of the form “we can compute sort(u) (or merge(v, w)) with at most so many comparisons,” and it seems that these statements correspond to statements of the form “the agent responsible for sorting u (or for merging v and w) in the recursive ASM has among its subordinates (i.e., vassals, vassals of vassals, etc.) at most so many merge-agents with nonempty inputs.”

A: Good. Perhaps one could also count, as evidence for our claim, that the program in Figure 1 looks very similar to Equations (1) and (2). Of course the syntactic details are different[6], but the updates (except for `Mode`) in the figure match what would be computed in evaluating the right sides of the equations.

[6]The notation of [6] can be improved to match the recursive equations more closely; such an improvement is incorporated in the ASM-based specification language AsmL [3]

6 Recursive ASMs as Distributed ASMs

Q: Right. But you also claimed that the runs of this recursive algorithm include the various implementations mentioned by Moschovakis, like sorting the left half first, or sorting the right half first, or interleaving the computations. So what, exactly, are the runs of a distributed algorithm?

A: That's a long story, written out in detail in [6, Subsection 2.2] and [5, Section 6]; the former converts recursive ASMs to distributed ASMs, and the latter defines runs of distributed ASMs. Rather than repeating the details here, let us just explain the essential ideas, with reference to the mergesort algorithm in Figure 1.

Converting a recursive ASM to a distributed ASM amounts to making explicit the creation of vassals, the waiting for the vassals' return values, and the use of these return values to continue the computation.

Q: This sequentialization will involve more modes, right?

A: Exactly. In [6] there was, in addition to the `Mode` involved in the recursive ASM, an additional `RecMode` in the distributed algorithm, whose purpose is to keep track of where in the "creating vassals, waiting for values, using the values" process an agent is.

Q: I suppose this `RecMode` is really a private `RecMode(Me)`, and that it has three possible values, corresponding to creating, waiting, and using.

A: You're right about `Me`, but in fact we need only two values for `RecMode`, since we can conveniently combine the waiting and using parts.

To make sure the idea is clear, let's describe what a sort-agent does in the non-trivial case where its input has length > 1. It starts with `Mode` $=$ `Initial` and `RecMode` $=$ `Create`[7] In this situation, it creates vassal sort-agents for the two halves of its input, and it goes into `RecMode` $=$ `Wait`. (By "creates" we mean that it imports two elements from the reserve and makes them vassals with the appropriate inputs.)

Q: How does it know to create the sort-agents corresponding to the lines $v :=$ `Sort`$(h_1(u))$ and $w :=$ `Sort`$(h_2(u))$ in the `rec Sort` block of Figure 1? Why doesn't it also create a merge-agent corresponding to the last line of the block?

[7]In [6] the values of `RecMode` were `CreatingSlaveAgents` and `WaitingThenExecuting`. We shorten them here.

A: Because it's in `Initial` mode, only the former lines are enabled, i.e., the guards governing them are true. The distributed program is arranged so that vassal agents are created only for the computations needed in enabled updates.

Q: This could get messy if an update involves nesting of recursively defined functions.

A: In [6], the mess was avoided by a convention forbidding such nesting. That's why Figure 1 has the updates involving `Sort` separate from those involving `Merge`, rather than having a single line

```
Result := Merge(Sort($h_1(u)$),Sort($h_2(u)$)).
```

Q: Moschovakis uses a similar disentangling of nested functions, though for a different reason, for example when he converts an abstract machine (or the corresponding tail recursion) to a recursor in [8, Section 4].

It seems clear that nesting of recursively defined functions can always be eliminated in this way.

A: Right. And if we didn't eliminate it in our recursive ASMs then the elimination would have to be incorporated into the conversion of recursive to distributed ASMs[8].

Let's get back to our sort-agent, which was last seen in `Mode` = `Initial` and `RecMode` = `Wait`. In this RecMode, the agent does nothing until its vassals are all in `Mode` = `Final`. Then it takes the vassals' `Result` values and puts them in place of the terms that had caused the creation of the vassals. Here this means that the vassals' `Result` values replace the terms `Sort`$(h_i(u))$. Now our sort-agent can execute the rest of its instructions. It assigns these values to v and w and enters `Mode` = `Halfway`. Having completed its `Wait` work, it also returns to `RecMode` = `Create`.

Since it's now in mode `Halfway`, the enabled update in its instructions is `Result` := `Merge`(v, w). So it creates a merge-agent with inputs v and w and enters `RecMode` = `Wait`.

Here it does nothing until its merge-vassal returns a value. Then, following its instructions, it assigns this value to `Result` and enters `Mode` = `Final` (and `RecMode` = `Create`).

Q: And here it does nothing, because there's nothing to do in `Final` mode.

[8]This would be the right approach, not artificially restricting recursive ASMs.

A: Right. `Final` mode terminates the computation. Should we also look at what a merge-agent does in the distributed computation?

Q: No, it's pretty clear in view of your explanation for sort-agents. Let's move on to the notion of runs.

7 Runs of Distributed ASMs

A: OK. There are actually two kinds of runs defined in [5, Section 6]. The simpler kind, sequential runs, are adequate to describe implementations of the sort you quoted Moschovakis as mentioning, so let's concentrate on these sequential runs first. Later, if we have time, we can look at the more general partially ordered runs.

A *pure sequential run* of a distributed ASM is a sequence of states S_n, in which the first state S_0 is an initial state of the ASM and each subsequent state S_{n+1} is obtained from its predecessor S_n by executing a move of an agent in S_n.

Q: That raises several questions. First, by "executing a move of an agent" do you mean performing the updates specified by that agent's program?

A: Yes. Recall that the agent's whole program describes a single computational step.

Q: Is the run a finite sequence or an infinite one?

A: It could be either. Of course for algorithms like mergesort, where a final answer is expected, the runs of interest would be the finite ones that end by giving `Output` a value (other than its original value of `undef`).

Q: Why is the word "pure" in the definition? What would an impure run be?

A: "Pure" means that all state changes come from the ASM, not from the environment. An impure run would allow intervention by the environment, for example a user typing input. Since we won't deal with impure runs here, we'll sometimes omit "pure."

Q: So a pure sequential run of a distributed ASM amounts to an arbitrary interleaving of the moves of the agents.

A: Right, but keep in mind that agents can be created and destroyed. So the possible moves from state S_n to state S_{n+1} depend on which elements of (the base set of) S_n are agents and which programs they have been assigned.

Q: OK. So for your mergesort ASM, there would be runs in which $h_1(u)$ is always sorted before $h_2(u)$, others in which they are sorted in the opposite order, and still others where the two sorting operations are interleaved, for example by having the merge-agents subordinate to these two sortings interleave their moves.

A: This is what we meant when we said that the runs of the mergesort ASM include implementations of the sort you quoted from Moschovakis.

Q: Do they include *all* implementations in Moschovakis's sense?

A: Answering that would require looking in detail at Moschovakis's definition of "implementation," a task that you suggested postponing.

Q: OK. That gives me some more motivation for understanding that notion of implementation.

Meanwhile, since pure sequential runs cover the obvious implementations, what's left for the more general partially ordered runs that you mentioned?

A: The idea here is that the relative order of some moves can be left unspecified. More precisely, a *pure partially ordered run*, or for short just a *run*, of a distributed ASM consists of

- a partially ordered set $(M, \leq)$ of (abstract) *moves*,
- a function A assigning to each move x an element $A(x)$ of the base set of the ASM's states (intuitively, $A(x)$ is the agent who makes move x), and
- a function σ assigning to each finite initial segment X of M a state $\sigma(X)$ of the ASM.

(An initial segment is just a downward-closed set.) These data are subject to the following requirements.

- For each $x \in M$, the set of predecessors $\{y \in M : y \leq x\}$ is finite.
- For each a, the subset $\{x \in M : A(x) = a\}$ of M is linearly ordered (by the ordering $\leq$ of M).

- $\sigma(\varnothing)$ is an initial state.
- If X is a finite initial segment of M, if x is a maximal element in X, and if $Y = X - \{x\}$, then $A(x)$ is an agent in the state $\sigma(Y)$, and $\sigma(X)$ is obtained from $\sigma(Y)$ by executing a move of this agent.

Q: That sounds pretty complicated. Can you give an intuitive explanation?

A: We can try. An element $x \in M$ represents a move made by an agent $A(x)$. (Part of the last requirement guarantees that when this move is made, $A(x)$ will be an agent rather than just some arbitrary element of the state.) The order relation $\leq$ on M refers to the relative timing of moves. $y < x$ means that move y must be completed before move x begins.

Q: When you say "must be," you leave open the possibility that y might be completed before x begins even if they are incomparable in M.

A: Right. Nothing is required about the relative timing of incomparable elements of M.

This is why the set $\{x \in M : A(x) = a\}$ of moves made by a single agent is required to be linearly ordered. Moves of a single agent occur in a definite order.

If X is a finite initial segment of M, then $\sigma(X)$ represents the state after the moves in X (and no others) have been made. So it certainly makes sense to require $\sigma(\varnothing)$ to be initial; it is in fact the starting state of the run.

The point of the final, complicated condition in the definition of run is that $\sigma(X)$ should be the state you reach by performing the moves in X in any order consistent with $\leq$. In particular, any maximal element x of X could be the last move in X. Just before that move, the state would be $\sigma(Y)$ where $Y = X - \{x\}$. So we require that, in this state, $A(x)$ is an agent, a move of which will lead to the state $\sigma(X)$.

Q: Now the definition makes more sense. What's the connection between sequential and partially ordered runs?

A: In the first place, every sequential run can be regarded as a partially ordered run in which the ordering $\leq$ of M happens to be a linear ordering. It's easy to check that, for linear $\leq$, the definition of partially ordered run essentially reduces to that of sequential run.

Secondly, if we have a partially ordered run and we increase the order relation, say from $\leq$ to $\leq'$ where $y \leq x$ always implies $y \leq' x$, then every initial segment with respect to $\leq'$ is also an initial segment with respect to $\leq$. So we can get a new run, using the same M and A, using $\leq'$ as the order, and restricting σ to the set of $\leq'$-initial segments — provided $\{y \in M : y \leq' x\}$ is finite as required in the definition.

In particular, $\leq'$ could be any linear extension of $\leq$ with finite initial segments. So, by linearizing the order, we can convert any partially ordered run to a sequential run.

Q: The linearization amounts to taking incomparable pairs, moves for which no relative timing was specified, in the partially ordered run, and arbitrarily imposing a relative order on them.

A: Right. Of course the "arbitrarily" is constrained by the demand that the result be an order. For example, if $y < z$ were both incomparable with x, then you couldn't put x before y and after z.

Q: Of course; I didn't intend "arbitrarily" to include absurdities, but I recognize the need to make intentions precise.

8 Algorithms

Q: Let me return to another issue that we barely mentioned earlier. You said that Moschovakis uses both the words *algorithm* and *machine* differently than you do. So far, we've discussed machines; the main difference seems to be that Mochovakis's machines operate in sequential time, while you allow for distributed computation. I'd be interested in hearing about the difference in meanings of "algorithm."

A: To localize the disagreement, let's first mention two points of agreement. First, there are some things that are obviously algorithms by anyone's definition — Turing machines, sequential-time ASMs, and the like. Even abstract machines in Moschovakis's sense are algorithms as long as the functions used by the machine (e.g., transition function and output function) are regarded as given.

Second, at the other extreme are specifications that would not be regarded as algorithms under anyone's definition, since they give no indication of how to compute anything. For example, the function sort could be specified by saying that sort(u) is the sequence with the same terms as u (with the same multiplicities) but in nondecreasing order. This can be regarded as a specification but it is too general to describe an algorithm. In fact, it is easy to give a declarative specification for uncomputable things, even though all the ingredients of the specification are computable. An example is the problem of deciding whether a given multi-variable polynomial with integer coefficients has an integer root.

Between the two extremes, there is a whole spectrum of things that give more or less information about how something is to be computed. The issue is how detailed this information has to be in order to count as an algorithm. Apparently, Moschovakis is more lenient here than we are, as far as recursion is concerned. He allows as algorithms some things that we would call only declarative specifications, and he would probably use the word "implementation" for the things that we call algorithms.

To be specific, let's look first at mergesort. Moschovakis says that the system of equations (1) and (2), or a recursor embodying these equations, defines an algorithm. We say that more information is needed before you have an algorithm. As already noted, the algorithm had better say that sort(u) is to be computed by the split-and-merge technique suggested by the equations, not by just any method that can be proved to satisfy the equations.

But there is more. The bottom-up approach you mentioned, where you sort all shorter strings before proceeding to any longer ones, is not among the approaches that Moschovakis's mergesort embodies, since it violates the bound on comparisons. Yet it certainly sorts strings by the split-and-merge technique suggested by the equations. Indeed, as you said, it's the algorithm implicit in the usual construction of least fixed points of monotone operators (like recursors).

Q: Isn't there another construction of a least fixed point, as the intersection of all sets closed under the operator?

A: Yes, and this leads to an even worse interpretation of the recursion equations — worse in the sense of computational feasibility.

It seems likely to us that, by the time one adds to recursion equations (or to a recursor) all the "fine print" needed to exclude unintended approaches to the computation (like the bottom-up approach or this intersection idea) that nevertheless flow from the equations in some sense, one has something that we'd call an algorithm. But we'd say that the algorithm is given not just by the recursor but by the combination of the recursor and the fine print. And we'd expect, in accordance with the ASM thesis, that this combination could be modeled by an ASM, in general a distributed one, as was the case for mergesort.

References

[1] Andreas Blass, Yuri Gurevich, and Saharon Shelah, "Choiceless polynomial time," *Ann. Pure Appl. Logic* 100 (1999) 141–187; addendum 112 (2001) 117.

[2] Andreas Blass and Yuri Gurevich, "Abstract state machines capture parallel algorithms," *ACM Trans. Computational Logic*, to appear.

[3] Foundations of Software Engineering Group, Microsoft Research, "AsmL Web Page," `http : //research.microsoft.com/foundations/#AsmL`.

[4] Yuri Gurevich, "Sequential abstract state machines capture sequential algorithms," *ACM Trans. Computational Logic* 1 (2000) 77–111.

[5] Yuri Gurevich, "Evolving algebras 1993: Lipari guide," in *Specification and Validation Methods* (ed. E. Börger), Oxford Univ. Press (1995) 9–36.

[6] Yuri Gurevich and Marc Spielmann, "Recursive abstract state machines," *J. of Universal Computer Science* 3 (1997) 233–246.

[7] Yiannis N. Moschovakis, "On founding the theory of algorithms," in *Truth in Mathematics* (ed. H. G. Dales and G. Oliveri), Clarendon Press, Oxford (1998) 71–104.

[8] Yiannis N. Moschovakis, "What is an algorithm?" in *Mathematics Unlimited* (ed. B. Engquist and W. Schmid), Springer-Verlag (2001) 919–936.