

Confusion of Memory

Lawrence S. Moss

Department of Mathematics
Indiana University
Bloomington, IN 47405 USA
February 14, 2008

Abstract

It is a truism that for a machine to have a useful access to memory or workspace, it must “know” where its input ends and its working memory begins. Most machine models separate input from memory explicitly, in one way or another. We are interested here in computational models which do *not* separate input from working memory. We study the situation on deterministic single-queue machines working on a two symbol alphabet. We establish a negative result about such machines: they cannot compute the length of their input. This confirms the intuition that such machines are “unable to tell” where on the queue the input ends and the memory begins. On the positive side, we note that there are some interesting things that one can do with such queue machines: their halting problem is undecidable, there are self-replicating machines, and there are recognizable languages outside of the control hierarchy.

Keywords: theory of computation, models of computation, formal languages

This paper is a comment on *input vs. memory/workspace* as they appear in machine models of computation. Usually, these are kept separate, using extra symbols, different tapes or registers, or other devices. Often in presentations this is done tacitly: “obviously” a machine would need to treat its input differently from the rest of its working space.

In this note, we wish to treat the matter in a little more detail, by proposing a simple computational setting where working memory is not clearly separate from input. One way to do this would be to look at Turing machines whose input might contain the blank symbol. Our proposal goes in a different direction, based on register machines (see Shepherdson and Sturgis [6] or other sources such as Enderton [1]). Normally, a register machine has many registers and works on numbers in unary notation, or else on words from some alphabet. We wish to consider only one register, and to make our point we must consider an alphabet A of at least two symbols. We take A to have exactly two symbols, 1 and $\#$. The input and workspace on such a machine comes in the form of a FIFO queue. The queue is unbounded, and all reading comes from one side and all writing to the other. There is no special “end of input” symbol, and there also are no other registers. So any writing that goes on the end of the queue will be conflated with the input.

The machine itself may be taken as a finite automaton controlling the queue. Alternatively, one can dispense with states and transition rules in favor of programs. Some of our results are more easily stated this way, and so we adopt a fixed language for programs.

A language for queue machine computations Programs are non-empty sequences of instructions, considered as sequences from A without recourse to coding, and instructions are of five types:

1#: add a 1 to the end of the queue.

1##: add a # to the end of the queue.

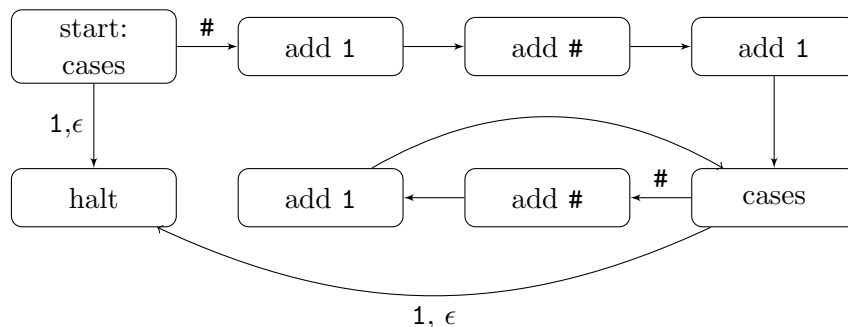
$1^k###$: skip forward k instructions.

$1^k####$: skip backward k instructions.

1#####: cases on the queue.

In the third and fourth types, $k \geq 1$. The case instruction works as follows: if the queue is empty, proceed to the very next instruction. If the queue begins with a 1, delete that entry and skip to the second instruction down. If the queue begins with a #, delete that entry and skip to the third instruction down. Running a program p on a word x in the queue is deterministic, and the run *halts* if control passes to an instruction “one below the bottom of p .”

An example Here is a very simple program, defined first in terms of a flowchart and then explicitly in our encoding. Let w be an input word w on the queue. If $w \in \{\#\}^*$, then the program is designed to output $(\#1)^{|w|}$. (That is, it replaces each # by #1 for inputs from $\{\#\}^*$.) For other inputs w , we don't care what the program outputs. One possible flowchart is shown below:



Each case statement has three branches, indicated by the labels on the edges from that instruction. The ϵ branch is executed if the queue is empty, and the 1 and # branches are executed if the queue begins with those symbols. The idea behind the chart is that if the input begins with a #, then the first case statement can learn this. At this point, we put a 1 on the end of the queue as an end-of-input marker. Then we repeatedly pop further elements off of the queue until we come to this end-of-input marker, at which time we halt. It is important that the case operations pop elements off front of the queue, and the write operations add to the end of the queue.

The flowchart is easily converted to an explicit program as we have defined it:

1#####	cases	1#####	cases
1 ¹¹ ###	go to the end	1 ⁵ ###	go to the end
1 ¹⁰ ###	go to the end	1 ⁴ ###	go to the end
1#	add 1	1##	add #
1##	add #	1#	add 1
1#	add 1	11111#####	go back 5

These twelve instructions then may be concatenated to yield a single word over $\{1, \#\}$. We take this to be the program:

1#####111111111111###111111111111###1#1##1#1#####11111###1111###1##1#11111####

For other examples of programs, see Proposition 5 and also [4].

Queue-computable partial functions Let $p, x, y \in A^*$. We write

$$\varphi_p(x) \simeq y$$

to mean that p is a program, and if p is run with x in the queue, then eventually the machine comes to a halt with y in the queue. For example, if p is the program written out in full before this discussion, we have $\varphi_p(\###) \simeq \#1\#1\#1$. We write $\varphi_p(\) \simeq y$ to mean the same thing, but when p is started on the empty queue.

Theorem 1 (Minsky [2, 3]). $\{p \in A^* : \varphi_p(\) \downarrow\}$ is Σ_1 -complete.

Proof Minsky's original proof was for Post's tag systems. The details here are a little bit different, but the ideas are close. The reduction is from Turing machines (TMs) on a one-letter alphabet $\{1\}$ which halt on the empty tape. For convenience, we need only consider TMs whose state set is a set of unary numerals. The easiest way to do this is to first reduce to register machines on $A = \{1, \#\}$ using six registers: one for the active state of a TM, one for the symbol under the reading head either the 1 or the blank symbol (here identified with #), one for the part of the tape to the left of the reading head, one for the part to the right, and two others to serve as temporary storage during copying. The central features of this reduction are the use of nested case statements to indicate the change of state and symbol under the head, and a program to find the last element of a queue while leaving the remainder of the queue unchanged. Once one has this reduction, a further reduction to a single queue is obtained by two modifications: First, increasing the alphabet, using a pair such as 11 for 1, another such as 1# for #, and then a third pair such as ## to separate blocks of other pairs. In this way, the contents of multiple registers (taken as words on the two-letter alphabet) may be taken to be single words on the the larger alphabet. The second modification is to consider adapt the basic operations which manipulate multiple registers to the new context, using circular copying. The details are for the most part straightforward. \dashv

To summarize: a queue machine with an empty queue can simulate a machine with more than one queue, and it can also simulate a machine working with a larger alphabet. One of the main points of this note is to point out that despite the Σ_1 -completeness of the halting set, the class of one-place functions which can be computed in our sense is quite limited.

Proposition 2. *Let $f : A^* \rightarrow A^*$ be the unary length map $f(w) = 1^{|w|}$. Then f is not queue-computable in our sense.*

Proof Suppose towards a contradiction that p were a program computing f . We write the instructions of p as i_1, \dots, i_N . Let

$$E_p = \{w : \text{the queue empties at some point in the run of } p \text{ on } w\}. \quad (1)$$

If the queue empties when we run p on w , it must be as a result of a case statement $1\#\#\#\#$. The reason is that these are the only instructions to delete symbols from the queue. For $w \in E_p$, let $n(w)$ be such that $i_{n(w)}$ is the instruction executed immediately after the queue empties for the first time when p is run on w .

Note that if $w \in E_p$, then $\varphi_p(w)$ is the same as the result of starting the program p on instruction $i_{n(w)}$ with the queue empty. In particular, this output only depends on $n(w)$. So if $w, x \in E_p$ and $n(w) = n(x)$, then $\varphi_p(w) = \varphi_p(x)$.

We next claim that there must be some string *not* in E_p . For suppose not. Then for each string w , $w \in E_p$ and $n(w)$ is defined; of course $n(w) \leq N$, since the program p has N instructions. Since there are infinitely many $w \in E_p$ and p is a (finite) program, there must be words w and x of *different lengths* such that $n(w) = n(x)$. By what we saw in the last paragraph, $\varphi_p(w) = \varphi_p(x)$. Hence $1^{|x|} = \varphi_p(x) = \varphi_p(w) = 1^{|w|}$. And so $1^{|w|} = 1^{|x|}$, which is to say $|w| = |x|$. We therefore contradict the assumption that w and x have different lengths. Our claim is proved.

Let $w \notin E_p$. This means that running p on w eventually results in $1^{|w|}$ on the queue, and the computation works in such a way that the queue is never completely empty. The original input w might be read, and at various points symbols s_1, \dots, s_M are added to the queue. What we mean is that $s = s_1 \cdots s_M$ is the sequence of all symbols, in order, which are added to the queue in the course of the computation of p on w . (It is even possible that $M = 0$; nothing in our argument requires $M > 1$.) The last $|w|$ symbols of s ($s_{M-|w|+1}, \dots, s_M$) are all 1s.

We consider two runs of p :

- (a) the run of p on w .
- (b) the run of p on $ws_1 \cdots s_M\#$.

As we know, the run of p on w eventually halts with $1^{|w|}$ in the queue. Our goal will be to show what happens in the run of p on $ws_1 \cdots s_M\#$. The point of the $\#$ at the end is to insure that this run converges with the queue containing a $\#$, but the import of this will not be clear until the end of this argument.

Let K be the number of steps in the computation of p on w . For $1 \leq i \leq K + 1$, let $m(i)$ be the instruction to be executed at step i . (For example, $m(1) = 1$. And for convenience, we set $m(K + 1) = N + 1$, where N is the number of instructions in p .) For $1 \leq i \leq K + 1$, let Q_i be the contents of the queue at the start of the i^{th} step. In general, Q_i is of one of two forms:

$w_r \cdots w_{|w|} s_1 \cdots s_j$ with $r < |w|$ and $0 \leq j \leq M$.

s_k, \dots, s_l with $k < l \leq M$.

The first case is a non-empty tail of w followed by an initial segment of s . (The initial segment of s might be empty.) The second case above is a non-empty subword of s .

Now Q_i and $m(i)$ are defined in terms of the run of p on w . We also define \overline{Q}_i and $\overline{m}(i)$ in the same way, but beginning with the input $ws_1 \cdots s_M \#$ rather than w .

Claim For $1 \leq i \leq K + 1$, the following hold:

1. $\overline{m}(i) = m(i)$, and
2. If $Q_i = w_r \cdots w_{|w|} s_1 \cdots s_j$, then $\overline{Q}_i = w_r \cdots w_{|w|} s_1 \cdots s_M \# s_1 \cdots s_j$.
3. If $Q_i = s_k \cdots s_l$, then $\overline{Q}_i = s_k \cdots s_l \cdots s_M \# s_1 \cdots s_l$.

The proof of this claim is by induction on i . For $i = 1$, $\overline{m}(i) = 1 = m(i)$, $Q_1 = w_1 \cdots w_{|w|}$ (so $r = 1$ and $j = 0$ in the second point), and $\overline{Q}_1 = w_1 \cdots w_{|w|} s_1 \cdots s_M \#$. Assume all three assertions of this claim for some i , and assume also that $i + 1 \leq K + 1$. If instruction $m(i)$ adds a 1 to the end of the queue, then $\overline{m}(i + 1) = \overline{m} + 1 = m(i) + 1 = m(i + 1)$, and $\overline{Q}_{i+1} = \overline{Q}1 = Q_i 1 = Q_{i+1}$. If instruction $m(i)$ adds a $\#$ to the end of the queue, the induction step is again easy. If instruction $m(i)$ is a forward or backward transfer, then the queue is irrelevant, and $\overline{m}(i + 1) = m(i + 1)$, and $\overline{Q}_{i+1} = \overline{Q}_i = Q_i = Q_{i+1}$. Finally (and this is the main point again), suppose that instruction $m(i)$ is a case statement, so it involves popping the first item from the queue and transferring according to whether the queue is empty, or if the popped symbol was a 1 or a $\#$. Then the queue is never empty for the run on w , and hence by induction hypothesis it is not empty for the run on $ws_1 \cdots s_M \#$. Again, we easily see that $\overline{m}(i + 1) = m(i + 1)$. We also argue that $\overline{Q}_{i+1} = Q_{i+1}$. The only non-trivial case is when (2) holds for i and $r = |w|$. So then Q_i is of the form $w_{|w|} s_1 \cdots s_j$, indicating that the queue contains only the last of the originally-input symbols. Hence Q_{i+1} is $s_1 \cdots s_j$. By induction hypothesis, the corresponding \overline{Q}_i is $w_{|w|} s_1 \cdots s_M \# s_1 \cdots s_j$. Executing instruction $m(i)$ leads to $s_1 \cdots s_M \# s_1 \cdots s_j$. This is point (3) in our claim, with $k = 1$ and $l = j$.

This concludes the proof of the claim. As a result, at the beginning of step $K + 1$, the run of p on $ws_1 \cdots s_M \#$ comes to a halt. By the claim, the contents of the queue at each point in the run is some sequence or other containing a $\#$. But since p computes the original function f of this lemma,

$$\varphi_p(ws_1 \cdots s_M \#) = \mathbf{1}^{|w|+M+1}.$$

And now we have our contradiction: on the one hand, $\varphi_p(ws_1 \cdots s_M \#)$ contains a $\#$. On the other hand, it is a sequence of 1 s. –

Similar reasoning shows that many functions are not queue-computable in this sense. For example: the reversal function, the doubling function $x \mapsto xx$, and the function $x \mapsto 1x$. In other words, one must use some encoding. Again, the point of this paper is to give a reason behind this “obvious” observation.

Self-replication with one queue Our next result is a contribution to the study of self-replicating computer programs. (These are more commonly known as “quines”.) We show that there are self-replicating queue machines in the first part of Proposition 3 below. However,

the slightest strengthening of this result fails, as we show in the next parts. (Note that for ordinary computation models such as Turing machines and register machines with more than one register, the Kleene Recursion Theorem insures that there are program p and q such that for all sequences x , $\varphi_p(x) = xp$, and $\varphi_q(x) = qx$.) So again, the interest here is in getting clear on just what can and cannot be done in this model of computation.

Proposition 3. *Concerning self-replicating programs on a single queue:*

1. *There is a program self such that $\varphi_{\text{self}}(\) = \text{self}$.*
2. *There is no program p such that for all sequences x , $\varphi_p(x) = xp$.*
3. *There is no program q such that for all sequences x , $\varphi_q(x) = qx$.*

Proof Before we begin the proof, we should mention that this result uses some special features of our encoding of queue machine programs. The first part uses the fact that there is a word $w \in A^*$ which is not a subword of any *program*, and the third part uses the fact that a program p without loops must have an output which is shorter than itself.

For the positive result in the first point, we consider functions $w, d : A^* \rightarrow A^*$ such that for all x ,

$$\begin{aligned} \varphi_{w(x)}(\) &\simeq x \\ d(x) &\simeq w(x) \cdot x \end{aligned}$$

For example, in our encoding we may take $w(1)$ to be $1\#$, since $\varphi_{1\#}(\) = 1$. Similarly, we may set $w(\#1) = 1\#\#1\#$, etc. The *idea* is to show that w and d are queue-computable, say by programs write and diag ; then to obtain the program self which we are after we need only consider $\varphi_{\text{diag}}(\text{diag})$.

However, the problem is that w and d are *not* queue-computable in our sense; again, the intuition is that a queue machine must be “confused about its input.” To get around this, we instead build a program write such that *for programs* q , $\varphi_{\text{write}}(q)$ is a program r with the property that $\varphi_r(\) = q$. Here is how to do this. Our encoding of programs by strings insures that $11\#1$ is not a subword of any program. So we begin write by putting $11\#1 \cdot 11\#1$ on the end of the queue. This allows us to simulate having three queues instead of one. (This move is similar to what we saw in our example near the beginning of the paper. We used a 1 as an “end-of-input marker.”) The first portion of the queue contains the initial input program q , and as the computation proceeds it will always contain a tail of q . The second portion begins empty, and as the computation proceeds it contains instructions to write an ever-increasing initial segment of q . The central point is to pick up the entries on the queue “four-by-four”, so that we know when to stop. Each time a symbol $s \in A$ is found in the first part, the machine copies the queue around until the second instance of the end marker $11\#1$ is found; at that point the instructions to write s are inserted before the end marker. (Again, those instructions are $1\#$ for $c = 1$ and $1\#\#$ for $c = \#$.)

The work for diag is similar, and so we omit the details. The point of diag is that for all programs q ,

$$\varphi_{\varphi_{\text{diag}}(q)}(\) \simeq \varphi_q(q).$$

This is because $\varphi_{\text{diag}}(q)$ is a program which, when run on the empty queue, will write q to the queue and then run q on this. Having diag , we do indeed get a self-replicating program $\text{self} = \varphi_{\text{diag}}(\text{diag})$, since

$$\varphi_{\text{self}}(\) \simeq \varphi_{\text{diag}}(\text{diag}) \simeq \text{self}.$$

We have omitted many routine details. We have explicitly given programs write , diag , self for our encoding of queue machines in [4]. We did not attempt to write the shortest programs, but our work carries out the sketch above. It turns out that diag contains 371 instructions, and self has 4,379. (self is so much longer because it consists of an instruction for each *symbol* of diag , followed by diag itself.) Running self on the empty queue converges with self as the output in 222,666,819 computation steps. (The most natural register machine program for diag would have *three* registers, and all of the work on the ending marker which we saw above is unnecessary. Done that way, the resulting self program has 202 instructions and replicates itself in 3,322 steps.)

Here is the proof of the second assertion in our result. It is similar to the argument for Proposition 2. Suppose towards a contradiction that $\varphi_p(x) = xp$ for all x . Let E_p be as in (1). Just as before, there must be some $w \notin E_p$. We claim that for this w (or indeed for any w), the program must read the entire input queue. For if not, write w as ab , where a is the subword of all symbols read as p is run on q , and b the part of w which is not read. Then for some c , $\varphi_p(w) = \varphi_p(ab) = bc$. By definition of p , $abp = bc$. If $a = \epsilon$, then p does not read even one symbol of its input $w = b$. (More precisely, the run of p on w (or any other input) does not contain any case statement $1#####$), yet in this run p itself is output onto the end of the queue after the input. Let m be the number of instructions in p . Then $|p|$, the length of p in symbols, is at least $2m$. (This is because in our encoding, the shortest instruction is 2 symbols long.) In particular, $m < p$. A program without case statements either has no loops, or if it has loops it does not converge. Hence the number of symbols written by p on any input is at most its number of instructions m . And since $m < p$, we see that p cannot write itself to the queue after its input. At this point we know that $a \neq \epsilon$. Let d be such that $|d| = |a|$ and $ab \neq bd$. Just as we saw before, $\varphi_p(abd) = bdc$. And by definition of p , $\varphi_p(abd) = abdp$. So the first $|a| + |b|$ symbols of $abdp = bdc$ are $ab = bd$, and this is a contradiction. So at this point, we know that every symbol of w is read on the queue in the computation of $\varphi_p(w)$. Again, let s be the sequence of symbols added to the queue in the computation of p on x ; by what we now know, wp is a tail of s . Then for all sequences t , $\varphi_p(wst) = wpst$. By definition of p , we have $\varphi_p(wst) = wstp$. Therefore $pst = stp$. However, this is impossible for all t : even for $t = 1$, we have a contradiction, since p cannot end in a 1.

We close with a sketch of the proof of the last part. Suppose towards a contradiction that q has the property that $\varphi_q(x) = qx$ for all x . As before, there is some $w \notin E_q$. Moreover, we may assume that the run of q on w reads all symbols of the input from the queue. Let s be the sequence of symbols added to the queue in the computation. So qw is a tail of s . For all sequences t , we have $qwst = \varphi_q(wst) = qwts$. So $st = ts$. But s must be non-empty, since every symbol of w is read from the queue. And if we take t to be the one-symbol sequence which differs from the first symbol of s , we contradict $st = ts$. \dashv

Language recognition A language $L \subseteq A^*$ is *queue-recognizable* if its characteristic function is φ_p for some $p \in A^*$; here we regard the elements of A as truth values, using 1 for *true* and #

We now prove that L is not context-free. We first observe that

$$L \cap \#^*1 = \{\#^{2^k-1}1 : k \in \omega\}.$$

To see this, note by the definition of L ,

$$\begin{aligned} \#^n 1 \in L & \text{ iff } \text{the } f(n+1)^{\text{st}} \text{ element of } \#^n 1 \text{ is a } 1 \\ & \text{ iff } f(n+1) = n+1 \\ & \text{ iff } n+1 \text{ is a power of } 2 \end{aligned}$$

In the last step, we used Lemma 4. Suppose toward a contradiction L were context-free. By the Pumping Lemma, let p be such that for every string $w \in L$ with $|w| \geq p$, there are u, v, w, y, z such that $w = uvxyz$, $|vxy| \leq p$, $|vy| \geq 1$, and for all i , $uv^i xy^i z \in L$. We apply this with $w = \#^{2^k-1}1$, where $2^k > p$. Let $r = |vy|$, so $1 \leq r \leq p$. Then $uv^2 xy^2 z = \#^{2^k-1+r}1$. Since

$$2^k - 1 < 2^k - 1 + r \leq 2^k - 1 + p < 2^{k+1} - 1,$$

the length of this pumped word $uv^2 xy^2 z$ is not a power of 2, and so $uv^2 xy^2 z \notin L$. This is a contradiction.

Similarly, the language L is not in the control language hierarchy, using the Pumping Lemma for that hierarchy due to Palis and Shende [5]. \dashv

At the present, we know very little about the queue-recognizable languages in our sense. It seems likely that all of them are recognizable in deterministic linear space, but we have not shown this. It also seems likely that languages such as the palindromes on a two letter alphabet will not be queue-recognizable.

Overall, our point here has been to clarify the situation regarding machines, input, and working memory. If a machine is liable to confuse its input with its working memory, then Proposition 2 shows that it might not be able to compute even very simple functions. Nevertheless, it might also be able to do other interesting things, as shown in Proposition 3 and 5.

Acknowledgment

I thank an anonymous referee for a suggestion which improved the presentation.

References

- [1] Enderton, Herbert B. **A Mathematical Introduction to Logic**. Second edition. Harcourt/Academic Press, Burlington, MA, 2001.
- [2] Minsky, Marvin L. Recursive unsolvability of Post's problem of "tag" and other topics in theory of Turing machines. *Ann. of Math.* (2) 74 1961 437–455.
- [3] Minsky, Marvin L. **Computation: Finite and Infinite Machines**. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1967.

- [4] Moss, Lawrence S. Recursion theorems and self-replication via text register machine programs, *Bulletin of the European Association for Theoretical Computer Science*, Number 89, 2006, 171–182.
- [5] Palis, M. A., and Shende, S. M. Pumping lemmas for the control language hierarchy. *Math. Systems Theory* 28 (1995), no. 3, 199–213.
- [6] Shepherdson, J. C. and Sturgis, H. E. Computability of recursive functions. *J. Assoc. Comput. Mach.* 10 217–255, 1963.
- [7] Weir, David J. A geometric hierarchy beyond context-free languages. *Theoret. Comput. Sci.* 104 (1992), no. 2, 235–261.