

A UNIVERSAL TURING MACHINE WITH 20 INSTRUCTIONS

MICHAEL SCHROEDER

ABSTRACT. We construct a universal Turing machine with five states, five symbols and 20 instructions in the standard model: one bi-infinite tape, left and right moves only, finite input on a blank background, and halting at an undefined transition. This improves on our earlier 21-instruction machine and other known machines in this model by Rogozhin and by Neary and Woods which use 22. The saving comes from a redesigned printing and restoration protocol that exploits a parity invariant. A direct compiler, with no intermediate tag system, drives the machine through a queue of unary addresses, periodically rebuilding its program so that the history it traverses stays short. It simulates a fixed source machine halting after t steps on input of length n in $O((t+1)(n+t+1)^2)$ time and $O((t+1)(n+t+1))$ space. For $n = O(t)$ this is cubic time, though the constants are astronomically large. Lean 4 proofs establish universality, halting equivalence, these bounds, and recovery of the source machine’s final configuration.

1. INTRODUCTION

We give a five-state, five-symbol universal machine with twenty instructions in the ordinary finite-input model, together with a compiler, resource bounds and an output reader. Rogozhin’s 4×6 machine and Neary and Woods’ 5×5 machine each use 22 instructions [17, 14]. Our machine uses two fewer; its state–symbol product remains 25. The comparison is with these explicit constructions. Historical model qualifications and the preceding unrefereed U_{21} preprint [18] are discussed in Section 11.

Three ideas connect the small table to general simulation and its time bound.

Parity replaces a restoration case. The interpreter temporarily cleans part of its program while executing a call. A rightward sweep restores pairs of cells, alternating between states C and E . Cleaned program runs have even length, whereas the history plus the consumed odd address has odd length. The sweep therefore reaches the consumed delimiter in state E : the instruction $E/X = gRA$ completes the final pair and resumes the next

Key words and phrases. universal Turing machine, instruction count, finite input, simulation complexity, formal verification.

ORCID: 0009-0004-3249-0195.

DOI: 10.5281/zenodo.23095173. © 2026 Michael Schroeder, CC BY 4.0.

call. This parity distinction underlies the coordinated printing/restoration change that saves one instruction relative to U_{21} .

Empty calls implement addition. An address is a run of b 's terminated by X . An empty production leaves one extra b at the unfinished end of the queue. Several empty calls therefore add an offset to the next output address. Encoding a preceding record by an even weight $\omega(u)$ and a current record by an odd base $\beta(v)$ makes $\omega(u) + \beta(v)$ select a table entry for the pair. Two passes of such lookups implement one source step.

Refresh keeps active history local. Every call crosses the history between its program and its queue. The exact cost in (4.1) is linear in that history length. The compiler therefore copies its program description while simulating the source, then installs a fresh bank beside the live data. Later calls cross only the new active history; retired banks remain on tape and count toward space. For a fixed source, a refresh interval (an epoch) spans a constant number of source steps and costs quadratically in the current window bound. Summing epochs gives the cubic-time, quadratic-space bounds when $n = O(t)$. The constants are enormous; the result concerns asymptotic overhead rather than a practical speed advantage.

The contributions are the twenty-instruction table, direct compilation with self-refresh, resource bounds, original-source readback, and their Lean verification for the same target run.

Theorem 1.1 (Universality and output recovery). *The machine U_{20} in Table 1 has 20 defined instructions. For every finite deterministic one-tape source T , start state q_0 and input x , a constructive finite encoding $\mathcal{E}_{T,q_0}(x)$ starts in state A at coordinate zero on a blank background, and*

$$T \text{ halts on } (q_0, x) \iff U_{20} \text{ halts from } \mathcal{E}_{T,q_0}(x).$$

From the actual target halt, a constructed finite-control tape machine produces an output representation of the source's final state, absolute head and tape at every integer coordinate. The construction requires no bound on future running time or tape use.

The following halting/readback conjunct is quoted from the final Lean theorem, with line breaks adjusted. Here R supplies the compiled program and reader; **Output** specifies the reader execution and interpreted output. Section 10 maps the full interfaces.

```
(Exec.Halts T.standardStep
  (T.denote (DirectStop.sourceInput T start xs)) ↔
  Exec.Halts U20.standardStep (U20.denote R.program.initial)) ∧
∀ n (d : StandardConfig s g),
  Exec.run T.standardStep n
  (T.denote (DirectStop.sourceInput T start xs))=some d →
  T.standardStep d=none →
  ∃ m target, 0 < m ∧
    standardRun m (U20.denote R.program.initial)=some (U20.denote target) ∧
    U20.standardStep (U20.denote target)=none ∧ R.reader.Output T target d
```

We abbreviate $\mathcal{E}_{T,q_0}$ by $\mathcal{E}_T$. The resource statement is Theorem 7.1. The *Technical supplement* supplies full interfaces, compiler details, optional optimizations and reproduction records.

2. MACHINE MODEL AND THE TWENTY INSTRUCTIONS

A source machine has finite nonempty state and symbol sets $\mathcal{S}, \Gamma$, a blank $\square \in \Gamma$, and a partial function

$$\delta_T : \mathcal{S} \times \Gamma \rightarrow \Gamma \times \{L, R\} \times \mathcal{S}.$$

A configuration is (q, p, τ) with $p \in \mathbb{Z}$ and $\tau : \mathbb{Z} \rightarrow \Gamma$. An instruction writes at p , moves one cell and updates the state; an undefined pair halts. Input x occupies $0, \dots, |x| - 1$, the head starts at zero, and all other cells are blank. Empty input represents one scanned blank.

The target states are A, B, C, D, E and its symbols are g, b, X, c, d , with blank c . An entry bLA means “write b , move left, enter A ”. All five undefined pairs are shown explicitly.

TABLE 1. The complete transition table of U_{20} . A dash is undefined.

Read	A	B	C	D	E
g	bLA	gRA	bLC	gRB	–
b	gLA	gRB	dRE	gRD	gRC
X	cRB	cRB	XRC	cRD	gRA
c	XLA	bLC	XLC	XLC	–
d	bLA	–	bLD	–	–

The row counts are 4, 5, 5, 4, 2, summing to 20. The undefined pairs are $(B, d), (D, d), (E, g), (E, c), (E, d)$. No stay-put move, extra halt state, second tape or external runtime action is implicit in this count.

2.1. Physical conventions. Words in the paper are written in increasing tape-coordinate order. A display $u[A]v$ puts the head on the first symbol of v in state A . Outside the displayed finite interval the tape is c . The Lean finite representation stores the left list nearest-cell first and the right list starting at the head; its denotation is an independent function $\mathbb{Z} \rightarrow \{g, b, X, c, d\}$. Simulation lemmas are lifted to this integer-tape semantics.

Unless a different level is explicitly named, target time counts defined transitions of Table 1. An undefined pair is the endpoint and contributes no extra transition. Source steps, queue calls and external reader steps are identified separately. Space is a bound on an interval containing both all visited head positions and every nonblank cell at every intermediate transition. This includes program banks, unary data, all retained history and the first exterior blank visited during a sweep. Initial *represented size*

counts the supplied finite lists, including designated blank cells. It is not a compressed file size or a count of nonblank symbols alone.

3. AN ADDRESS QUEUE ON ONE TAPE

For a list of addresses $Q = (n_1, \dots, n_k)$ and an unfinished tail $z \geq 0$, define

$$(3.1) \quad D(Q, z) = b^{n_1} X \dots b^{n_k} X b^z.$$

The tail is stored on tape as a run of b 's without a terminating X . At an ordinary checkpoint the tape has the form

$$(3.2) \quad \mathcal{A} P (dg)^h [A] D(Q, z),$$

where P is the current program, h counts history *pairs*, and $\mathcal{A}$ is an arbitrary finite archived prefix. Initially the archive is empty. Ordinary callable addresses are positive and odd. Protected descriptor fields used during installation may be even; they are not dispatched as ordinary calls.

Suppose the first address n selects a production $V(n) = (a_1, \dots, a_r)$. For $Q = (n)R$, the queue action is

$$(3.3) \quad \begin{aligned} ((n)R, z) &\mapsto (R, z + 1), & r = 0, \\ ((n)R, z) &\mapsto (R(z + a_1) a_2 \dots a_r, 0), & r > 0. \end{aligned}$$

The history pair count becomes $h + (n + 1)/2$. The consumed address and its delimiter therefore become exactly $n + 1$ additional history cells.

An empty production does useful work: it increments the tail. A later nonempty output appends to that tail before printing its first delimiter. Thus address addition is concatenation of adjacent unary runs. No arithmetic transition is added to Table 1. Because the next address depends on the accumulated tail, this mechanism is not a fixed-production one-symbol deletion system.

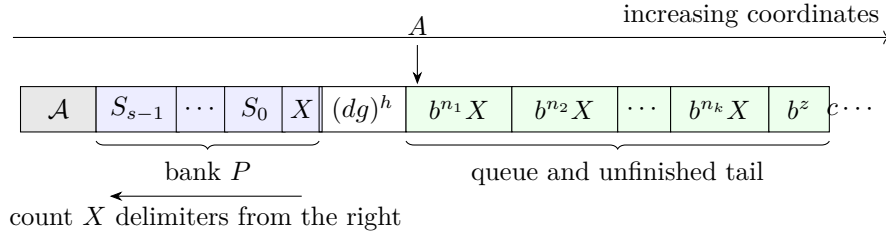


FIGURE 1. An ordinary checkpoint. The head scans the leading address. Slot order is reversed on tape; slot 0 is nearest the history. Fixed header cells are suppressed. Widths are schematic.

3.1. Address addition in a queue example. Consider the three productions

$$V(1) = (), \quad V(3) = (3), \quad V(5) = (3, 9).$$

Start with queue $(1, 1, 3)$, tail zero and no history. Table 2 shows five completed calls, up to the arrival of address 9 at the front. The next subsection constructs the bank and explains its halt at that address.

TABLE 2. Five completed calls in the queue example. History is measured in pairs; costs count target transitions for the bank constructed below.

Completed calls	Queue	Tail	History pairs	Cost
0	$(1, 1, 3)$	0	0	–
1	$(1, 3)$	1	1	24
2	(3)	2	2	30
3	(5)	0	4	208
4	$(3, 9)$	0	7	1,510
5	$(9, 3)$	0	9	412

Two empty calls turn the copied address 3 into 5, selecting the production $(3, 9)$.

3.2. Finite slots and addresses. A *slot* is a program entry containing a sequence of output *frames*, each of which prints one terminated unary field. Slot j has an even delimiter width $w_j = 2v_j$ and a list of frame parameters $(\kappa_{j,1}, \dots, \kappa_{j,r_j})$, with $r_j + 1 \leq w_j$. Put

$$F(k) = X(dg)^{k+1}, \quad S_j = X^{w_j-r_j} F(\kappa_{j,r_j}) \cdots F(\kappa_{j,1}).$$

The finite standalone program and its entry addresses are

$$(3.4) \quad P = dd S_{s-1} \cdots S_0 X, \quad n_j = 1 + \sum_{i < j} w_i.$$

Every slot has exactly w_j delimiters, and every n_j is odd. Addressing depends on delimiter counts, not on the unary lengths inside frames. The header address is $1 + \sum_{i < s} w_i$.

A frame $F(k)$ prints $b^k X$. An additional b is printed by the common entry sweep. Hence a nonempty frame list implements

$$V(n_j) = (\kappa_{j,1} + 1, \kappa_{j,2}, \dots, \kappa_{j,r_j}).$$

The empty frame list implements the first line of (3.3). Compilers enforce positivity and parity for fields that will be called. Generalized banks replace the standalone header by a retained archive and allow explicit padding runs; the same local proof applies to their live slots.

The initial physical size of the standalone bank is exactly

$$(3.5) \quad |P| = 3 + \sum_{j < s} \left(w_j + 2 \sum_{i=1}^{r_j} (\kappa_{j,i} + 1) \right).$$

This illustrates why small instruction count does not force a small compiled program: an address is represented in unary and can occur in many frames.

For the example in Section 3.1, take three slots with widths 2, 2, 4 and frame lists $()$, (2) , $(2, 9)$. Their addresses are 1, 3, 5, and the header address is 9. Equation (3.5) gives a 43-cell bank. The initial data $D((1, 1, 3), 0) = bXbXb^3X$ occupy eight cells, giving a complete 51-cell input with no history. After the five calls in Table 2, the header dispatch costs 815 transitions and halts at (D, d) : **2,999** transitions in all. With the program starting at coordinate zero, the head moves from 43 initially to zero at halt. Lean kernel reduction and an independent interpreter agree on the whole final configuration.

4. INDEXING, TWO-CELL PRINTING AND RESTORATION

Lemma 4.1 (Complete ordinary call). *From a canonical configuration whose leading odd address selects a live slot, U_{20} executes a positive finite sequence of defined transitions and reaches the canonical configuration prescribed by (3.3). The program and archived prefix are restored exactly. Every intermediate head position and nonblank cell satisfies the corresponding physical interval bound.*

Proof. There are three phases. A *token* is the two-cell pattern used to append one output symbol. Its *corridor* is the traversed tape between that pattern and the output end; a *clean corridor* contains only b and X . The following invariants govern each phase.

Indexing. Each b of the leading address causes an outward and return sweep that crosses one further program delimiter. In state A , g and d are cleaned to b , and temporary c marks are restored to X . The next untouched X turns the head right in state B . The rightward sweep marks traversed b 's as g and delimiters as c ; the newly marked data cell returns control to A . After n rounds, the machine has located the n th program delimiter counted from the right. It traverses the remaining data, appends one b , and returns in C to the selected frame. The crossed corridor is now clean, containing only b, X .

Printing. With C on the right-hand d of gd , the instruction $C/d = bLD$ puts D on g . The instruction $D/g = gRB$ starts a rightward sweep which appends b ; the C return cleans the token to bb . With C on the d of Xd , the D/X branch instead appends X and cleans the token to Xb . For corridor length ℓ_{cor} , either token costs $2\ell_{\text{cor}} + 6$ transitions. Reading a frame from right to left first skips g , then executes k tokens gd and a final Xd . For

example,

$$F(2) = \underbrace{Xd}_{\text{print } X} \underbrace{gd}_{\text{print } b} \underbrace{gd}_{\text{print } b} \underbrace{g}_{\text{skip}} \leftarrow C$$

prints bbX . Reversing the frames gives forward output order.

Restoration. A stop delimiter enters the rightward restoration sweep. On each cleaned pair bb , the instructions

$$C/b = dRE, \quad E/b = gRC$$

write dg and return to C . Every cleaned program run has even length, so $C/X = XRC$ preserves each internal delimiter. Beyond the program, the cleaned history and consumed address have length $2h+n$, which is odd. The sweep therefore reaches the consumed delimiter in state E . The instruction $E/X = gRA$ completes the new history pair and exits on the next data cell in A . All untouched cells remain unchanged. $\square$

4.1. An exact local cost. Let $V_j = \sum_{i < j} v_i$ be the half-width preceding slot j , μ its frame-output length, excluding the common initial b , and ℓ_{suf} the tape length after the selected address. For history count h , the exact cost is

$$(4.1) \quad T_j(h, \ell_{\text{suf}}) = T_j(0, 0) + 4(2V_j + \mu + 2)h + (2 + 2\mu)\ell_{\text{suf}}.$$

The finite slot layout determines $T_j(0, 0)$. The proof constructs a run of that length. In particular, the history term explains why refresh is needed: every later call otherwise pays for earlier history. Figure 2 shows the smallest call in the fixture. It has an empty production, so its only appended symbol is the common initial b ; no frame-printing token is executed.

step	41	42	43	44	45	46	47	48	49	50	51	state	operation	
0	X	X	b	X	b	X	b	b	b	X	c	A	index address 1	
1	X	X	g	X	b	X	b	b	b	X	c	A		
2	X	c	g	X	b	X	b	b	b	X	c	B		
3	X	c	g	X	b	X	b	b	b	X	c	A		
4	X	c	g	c	b	X	b	b	b	X	c	B	sweep right; append <i>b</i>	
5	X	c	g	c	g	X	b	b	b	X	c	B		
6	X	c	g	c	g	c	b	b	b	X	c	B		
7	X	c	g	c	g	c	g	b	b	X	c	B		
8	X	c	g	c	g	c	g	g	b	X	c	B		
9	X	c	g	c	g	c	g	g	g	X	c	B		
10	X	c	g	c	g	c	g	g	g	c	c	B		
11	X	c	g	c	g	c	g	g	g	c	b	C	return; clean data	
12	X	c	g	c	g	c	g	g	g	g	X	b		C
13	X	c	g	c	g	c	g	g	b	X	b	C		
14	X	c	g	c	g	c	g	b	b	X	b	C		
15	X	c	g	c	g	c	b	b	b	X	b	C		
16	X	c	g	c	g	X	b	b	b	X	b	C		
17	X	c	g	c	b	X	b	b	b	X	b	C		
18	X	c	g	X	b	X	b	b	b	X	b	C	restore; $E/X = gRA$	
19	X	c	b	X	b	X	b	b	b	X	b	C		
20	X	X	b	X	b	X	b	b	b	X	b	C		
21	X	X	b	X	b	X	b	b	b	X	b	C		
22	X	X	b	X	b	X	b	b	b	X	b	C	next checkpoint	
23	X	X	d	X	b	X	b	b	b	X	b	E		
24	X	X	d	g	b	X	b	b	b	X	b	A		

FIGURE 2. Every configuration of the first fixture call: 24 transitions and 25 rows. Columns are tape coordinates; the shaded cell is the head. All cells outside $41, \dots, 51$ stay unchanged. The production is empty, so there is no frame-printing phase. At step 23, $E/X = gRA$ completes the history pair; step 24 has queue $(1, 3)$ and tail 1.

5. DIRECT SIMULATION OF AN ORDINARY MACHINE

The compiler represents a source window as

$$(5.1) \quad K \text{ Plain}(a_\ell, o_\ell) \cdots \text{Head}(q, a_p, o_p) \cdots \text{Plain}(a_r, o_r) \text{ G},$$

where o_i is an origin bit. Exactly one cell is headed and exactly one cell has its origin bit set. The origin remains attached to its source coordinate when the symbol is rewritten. New exterior blank cells have origin bit zero. This permits absolute, rather than only head-relative, output recovery.

Each source transition is implemented by two adjacent-record passes, called *forward* and *return*. Their alphabets are finite for fixed T . Intermediate records include $\text{Left}(q, a, o)$, $\text{Right}(q, a, o)$, $\text{Frozen}(q, a, o)$ and a boundary record $\text{Kleft}(q)$. The following cases describe their computational content.

5.1. Writing and moving the head. For $\delta_T(q, a) = (a', R, q')$, the forward pass changes the headed cell to $\text{Right}(q', a', o)$, leaving other symbols unchanged. The return pass makes that cell plain and transfers the head state to its successor. If the successor is G , the pass emits a new blank headed cell followed by G .

For $\delta_T(q, a) = (a', L, q')$, the forward pass writes a' at the old head and marks the predecessor with $\text{Left}(q', \cdot, \cdot)$. The return pass turns that predecessor into the new headed cell. At the left frontier, the forward pass replaces K by $K\text{left}(q')$ and the return pass emits $K\text{Head}(q', \square, 0)$. No preallocated future cells are required.

If $\delta_T(q, a)$ is undefined, a frozen record retains the source state and symbol. The mathematical return pass can recover exactly the same source configuration; the executing compiler detects this event and enters a terminal snapshot protocol. It does not simulate an additional defined source step.

Lemma 5.1 (Two-pass source invariant). *For every valid marked source configuration κ and every defined source step $\kappa \rightarrow \kappa'$, the complete forward/return cycle transforms the guarded encoding of κ into that of κ' . Both source boundaries may grow. The represented window grows by at most one cell per source step, and the unique head and origin invariants are preserved.*

Proof. The four cases are left and right moves, each with an existing neighbour or a frontier. The displayed rules write and transfer the head correctly in each case. All other cells retain their marks; a fresh cell has origin bit zero. The marked-tape invariant lifts this window equality to the entire integer tape. $\square$

5.2. A source step through both passes. Let a source write 1, move right and enter a halting state q_1 from $(q_0, 0)$. With 0^* marking the origin, the logical trace is

$$\begin{aligned} K\text{Head}(q_0, 0^*) \text{Plain}(0) G &\xrightarrow{\text{forward}} K\text{Right}(q_1, 1^*) \text{Plain}(0) G \\ &\xrightarrow{\text{return}} K\text{Plain}(1^*) \text{Head}(q_1, 0) G. \end{aligned}$$

The next undefined action triggers terminal processing. The reader recovers q_1 , head coordinate 1, a 1 at zero and blanks elsewhere. With no second input cell the rule creates it at G . Section 9 reports the compiled replay.

5.3. Adjacent lookup by pulse addition. The queue must implement a finite function of two adjacent records, although a live call selects only one address. Assign an even weight $\omega(u)$ to the preceding record and an odd base address $\beta(v)$ to the current record. A *pulse* is an empty call; a *carrier* is a call that emits the base address. A synchronized expansion places the pulses and carriers in the required queue order. Empty calls accumulate $\omega(u)$ in the unfinished tail; the next carrier prints $\beta(v)$, so the completed address is

$$(5.2) \quad \omega(u) + \beta(v).$$

The bank entry at that odd address contains the required two-record output. The reference layout makes this formula explicit. Number the forward-phase records by $0, \dots, M_A - 1$, placing a distinguished end-of-pass record (the

sentinel) first. Every lookup slot has six delimiters. For previous/current indices i, j , its address is

$$6M_A i + 6j + 5 = \underbrace{6M_A i}_{\omega(i)} + \underbrace{(6j + 5)}_{\beta(j)}.$$

The return-phase matrix follows it, with addresses $6M_A^2 + 6M_B i + 6j + 5$. Quotient and remainder by the row width distinguish the pairs, all addresses are odd, and the sentinel weight is zero. Helper and control regions follow these matrices.

For an illustrative 2×2 forward lookup block, the addresses are

	$j = 0$	$j = 1$	
$i = 0$	5	11	$\omega(1) = 12, \quad \beta(0) = 5.$
$i = 1$	17	23	

Let e denote an empty call and C_0 a carrier with output (5). Once the helper expansion has placed twelve pulses before that carrier, the queue action is

$$((e^{12}, C_0)R, 0) \mapsto^{12} ((C_0)R, 12) \mapsto (R(17), 0).$$

Here e^{12} means twelve queue entries, not a unary tape field. The resulting address selects row $i = 1$, column $j = 0$. This calculation also shows why timing matters: the pulses must reach the unfinished tail immediately before the intended carrier.

Finite graphs of intermediate *helper* calls split the weights and synchronize the pulses. Whole-pass invariants specify the queue and its tail. The *Technical supplement* details the layers and offset cancellation.

6. AUTONOMOUS REFRESH OF THE ACTIVE PROGRAM

The compiler keeps a *template*, an encoded description of its program, beside the source window. A concurrent *copier* duplicates that template using the same adjacent-record passes as the source simulation. A template of length λ completes an epoch after

$$(6.1) \quad d_{\text{ep}} = \lambda + 3$$

source steps. This period depends only on the compiled source table.

The first cycle duplicates each seed a as $L(a)R(a)$ and adds $R(\perp)$ at the front and $L(\perp)$ at the back. Subsequent cycles swap every adjacent R, L pair simultaneously, preserving the order within each tag class. Such pairs are disjoint. After $\lambda + 1$ sorting cycles the left copy precedes the right copy. A final cycle turns $L(\perp)$ at the adjacent $L(\perp)R(\perp)$ boundary into the unique completion event, leaving $R(\perp)$ in place. Thus duplication, sorting and completion take $1 + (\lambda + 1) + 1 = \lambda + 3$ cycles.

For $\lambda = 2$, the complete schedule is below. Fixed outside markers are suppressed, and **done** is the completion event.

cycles	copier region
0	$\text{seed}(a) \text{ seed}(b)$
1	$R(\perp) L(a)R(a) L(b)R(b) L(\perp)$
2	$L(a) R(\perp)L(b) R(a)L(\perp) R(b)$
3	$L(a)L(b) R(\perp)L(\perp) R(a)R(b)$
4	$L(a)L(b)L(\perp) R(\perp)R(a)R(b)$
5	$L(a)L(b) \text{ done } R(\perp)R(a)R(b)$

The formal completion theorem also excludes an earlier event. Source updates proceed concurrently in the disjoint guarded region.

6.1. Constructing a bank that describes itself.

Remark 6.1 (Delimiter addresses break the apparent circularity). Slot addresses depend on delimiter counts, while unary lengths inside frames determine physical size. The compiler can therefore fix all addresses from production arities before filling in the data that describes those addresses. Enlarging an encoded unary run leaves the assigned addresses unchanged.

This is the essential self-description step. The compiler enumerates source records, lookup rows and helper nodes in disjoint regions, then resolves their symbolic references. A unit template denomination makes every descriptor run representable. The reference choice uses one unit letter and binary expansion of depth one to reconstruct its run. These finite constructions produce a refresh certificate for every source table; adding the snapshot terminal gives the compiler used in Theorem 1.1.

6.2. Installing the copied program. A *descriptor* is a protected sequence of even fields encoding the physical runs of a bank. A promotion call turns those runs into executable program cells, archives the old bank and its history, and resumes beside the new bank. A short *bootstrap* routine absorbs the known unfinished-tail offset.

Refresh uses two installations: from the active bank P to an *adapter* P_{aux} , then back to a fresh copy of P . The adapter preserves source records while its *serializer* converts copied template letters into the descriptor of P . Every descriptor and every new bank is written by the twenty-instruction machine itself.

Lemma 6.2 (Complete epoch). *A source run of d_{ep} defined steps from a valid κ to κ' induces a positive target run from the canonical encoding of κ to a fresh canonical encoding of κ' . Installation preserves the source. Returned active history is bounded by a compiler-dependent constant, and the new archive contains the old archive unchanged.*

Proof. The counted source passes produce the finished copier together with κ' . The round trip installs P_{aux} , converts the template to the descriptor of

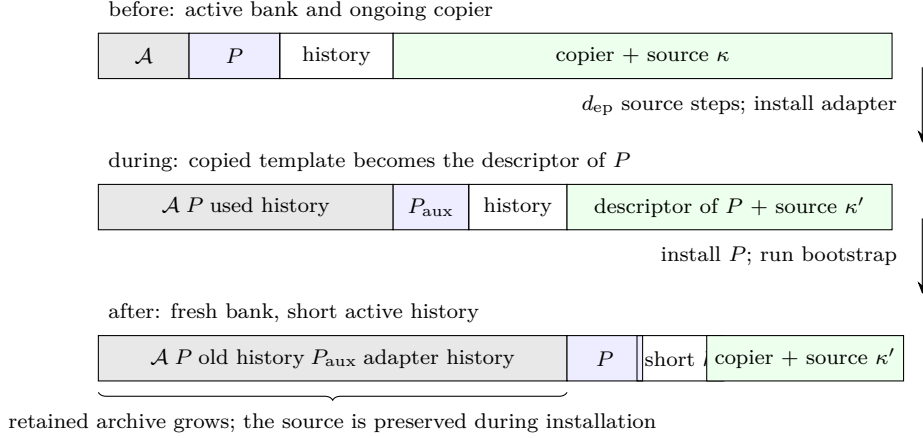


FIGURE 3. One refresh epoch. Bank and queue widths are schematic; fixed separators, transient fields and bootstrap calls are suppressed.

the entire bank P , installs it, and executes its bootstrap. The tail-offset invariant identifies the resumed queue with the canonical source queue. The archive constructor retains both retired banks and their histories; induction retains all earlier archives. $\square$

6.3. Halting inside an epoch. The source may halt while the copier is unfinished. A terminal routine saves a *snapshot* of the mixed queue, whose records may occupy different helper layers. Its decoder removes recognized helpers and recovers the common source layer. Terminal processing reaches an undefined target pair in finitely many transitions, including for a source with no defined transition.

Conversely, each valid live phase and installation has a defined, finite execution. Only an undefined source action starts the terminal routine. Iterating the positive epoch simulations of a nonhalting source rules out any finite target halt. This proves both directions of halting equivalence, including halts between epoch boundaries.

7. SIMULATION BOUNDS

All constants in this section are computed from the chosen finite reference compiler for the fixed source T . We write $n = |x|$, $N = \max(1, n)$, and let t count defined source transitions before halting.

Theorem 7.1 (Reference resource bounds). *For the encoding in Theorem 1.1, there are explicit computable constants B_T, I_T, K_T, J_T, S_T , depending only on T and its blank, uniformly over start states and inputs, such that:*

(i) $|\mathcal{E}_T(x)| \leq I_T(n+1)$;

(ii) *a source halt after t steps yields the target halt in $m > 0$ transitions, with*

$$(7.1) \quad m \leq (t+1)K_T(B_T + N + t)^2;$$

(iii) *throughout that same run, all target head positions and all nonblank cells lie in an interval of at most $S_T(t+1)(N+t+1)$ cells, including all archived tape.*

The halted configuration is the one read by the output theorem.

7.1. From local effects to the complete bound. A sequence of calls and installations is assigned a natural weight w . Starting with history h and q queue cells, it takes at most

$$(7.2) \quad 30w(1 + h + q + w)$$

transitions, returns active-history-plus-queue potential at most $h+q+w$, and grows the right boundary by at most w while preserving the left boundary. Weights compose by addition: after weight a , weight b costs at most $30b(1 + h + q + a + b)$, so the combined time is bounded by $30(a+b)(1 + h + q + a + b)$. These statements include every intermediate transition and all archived cells.

The source window has at most $N + t$ cells. For $W = B_T + N + t$, the finite alphabets and helper graphs give at most $C_q W$ queue cells and weight $C_e W$ per epoch, including the copier and both installations. Each epoch starts and ends with history at most a fixed H_0 . Equation (7.2) therefore gives $O_T(W^2)$ time and $O_T(W)$ interval growth per epoch. This reset keeps the cost local even though the archive continues to grow.

Write $t = kd_{\text{ep}} + r$, $0 \leq r < d_{\text{ep}}$. After k complete epochs, a terminal segment handles the remaining r source steps, snapshot preparation and target halt. Its bounds have the same forms, $K_{\text{term}}W^2$ and $J_{\text{term}}W$. Since $k \leq t$, summation gives

$$(7.3) \quad \begin{aligned} m &\leq (t+1)K_TW^2, \\ \text{left endpoint} &= -F_T, \\ \text{right endpoint} &\leq \|Q_0\| + (t+1)J_TW, \end{aligned}$$

where F_T is the bank length and $\|Q_0\|$ the initial queue length in tape cells. Thus a containing interval has at most

$$(7.4) \quad F_T + \|Q_0\| + (t+1)J_TW + 1$$

cells, counting both endpoints. Deterministic terminal uniqueness identifies this execution with the readback witness.

The input construction gives

$$(7.5) \quad |\mathcal{E}_T(x)| = F_T + \|Q_0\| \leq F_T + C_q(B_T + N) \leq I_T(n + 1),$$

where $I_T = F_T + C_q(B_T + 1)$. One may take

$$S_T = I_T + 1 + J_T(B_T + 1).$$

Indeed, $n + 1 \leq N + t + 1$ and $W \leq (B_T + 1)(N + t + 1)$ turn (7.4) into $S_T(t + 1)(N + t + 1)$. This proves Theorem 7.1, including $t = 0$. With $n = O(t)$ the upper bounds are cubic time and quadratic retained space. The *Technical supplement* supplies the exact coefficient definitions.

Proposition 7.2 (Time to encode a fixed source). *For fixed T , the reference input can be produced by a deterministic streaming transducer with separate input and output tapes in $O_T(n+1)$ time and $O_T(1)$ working space, excluding its output.*

Proof. Emit the fixed program bank and auxiliary prefix. For empty input, emit a blank head record carrying the origin mark. Otherwise mark the first input letter as the head and origin, encode the remaining letters as unmarked plain records, and append the fixed guards. Each record emits a unary block from a finite dictionary depending only on T . A finite-control scan therefore writes a bounded number of cells per letter. Constructing the fixed dictionary and bank, including its emission order, contributes a T -dependent constant. $\square$

This proposition concerns a streaming encoder for a fixed source table. The costs of the Lean evaluator and of constructing a compiler for a varying source table are separate questions.

8. PHYSICAL RECOVERY OF THE ORIGINAL OUTPUT

Halting equivalence alone does not specify what the halted tape means. The reference terminal routine leaves a dispatch snapshot from which a finite reader recovers the entire source output. Its inputs are the fixed compiled tables and the actual target tape.

The first reader pass scans the saved prefix and emits an address tape. The first saved field includes a history contribution; a fixed modulus $2n_{\max} + 4$, where $n_{\max}$ bounds the compiled addresses, separates the admissible address from that contribution. The finite-state implementation stores only a bounded residue and a saturated counter.

The second pass uses a fixed inverse dictionary, removes helpers, and emits the marked source sides and state/current metadata. The origin mark determines the absolute head. Interpreting this finite output recovers the source state and tape at every integer, including both blank exteriors.

The reader starts at the target's actual halted head. Its coordinate axis is reflected, so a right move of its input head is a left move on the target tape. The proved terminal shape makes the interpreted input tape exactly $\tau_{\text{target}}(-z)$ at every integer z . This is a change of orientation, requiring no

copying of the saved prefix and no unbounded coordinate arithmetic by the reader.

Proposition 8.1 (Direct physical reader). *For a genuine halted direct compilation, let ℓ_{saved} be its saved-prefix length, r_{saved} its saved address count, and ℓ, r its final source-side lengths. The finite-control five-tape reader takes exactly*

$$\ell_{\text{saved}} + r_{\text{saved}} + 4$$

reader transitions and uses at most $r_{\text{saved}} + \ell + r + 6$ workspace cells at every prefix. For fixed bank parameters,

$$\ell_{\text{saved}} \leq r_{\text{saved}}(2n_{\text{max}} + 5) + \text{boot} + 2\text{tailBound} + 20.$$

The five tape roles include the read-only U20 tape. Workspace excludes that input and the fixed reader transition table, and retains the intermediate address tape and all output buffers. U20’s separate physical-space bound already includes the input tape. This separate reader starts from the same target halt bounded in (7.3).

9. CONCRETE COMPILED INSTANCES

Independent compressed queue replays complete the reference compiler’s output for two source machines and recover their final configurations. The *zero-transition source* has one state, one symbol and no defined transition; its input is empty. The *one-step writer* has two states and two symbols. In state q_0 it writes 1, moves right and enters q_1 , where both actions are undefined. Its input is 00, as in Section 5. In both compilations the designated blank is used for auxiliary filler records.

TABLE 3. Actual reference sizes and completed compressed queue replays. Transition totals are calculated by summing exact local costs along the replay, not by executing every target transition.

Quantity	Zero-transition source	One-step writer
Source steps t	0	1
Bank cells F_T	1,202,566,981	5,940,353,221
Template letters λ	601,250,868	2,970,096,575
Initial cells $F_T + \ Q_0\ $	7,165,709,939,107	78,755,085,930,851
Queue calls	775,644,398,417	36,039,771,581,614
Calculated target transitions m	2.527899×10^{29}	2.644035×10^{32}
Halt and decoded output	verified	verified

The simulator consumes the production tables exported from Lean, representing long repetitions of addresses in compressed blocks. It preserves unfinished-tail arithmetic and constructs the terminal bank by parsing the descriptor actually produced by the queue. Both runs reach the undefined

pair E/c . It saves the compressed terminal tape, reloads it, and applies the snapshot and source decoders. The first output is state 0, head 0 and all blanks; the second is state 1, head 1, symbol 1 at coordinate 0 and blanks elsewhere. Expected outputs are compared only after the run and readback have completed.

These examples expose the scale hidden by the asymptotic notation. For the zero-transition source, (7.1) gives approximately 3.695×10^{132} , about 1.462×10^{103} times the calculated halt count. Its initial-size coefficient is likewise loose: $I_T/(F_T + \|Q_0\|) \simeq 3.143586 \times 10^{10}$. The table therefore reports actual initial cells alongside the completed replay costs.

Both runs halt before their first periodic refresh. The periods are 601,250,871 and 2,970,096,578 source steps. The runs exercise terminal-bank installation. A separate 7,320-transition literal fixture checks a small supplied-descriptor bank-refresh round trip. The *Technical supplement* gives the differential checks, million-transition literal prefixes, all coefficients and the scope of the compressed reader.

10. FORMAL VERIFICATION

The development uses Lean 4.32.2 and its standard library [6]. The partial machine table refines an independent transition function on tapes $\mathbb{Z} \rightarrow \{g, b, X, c, d\}$. The public theorem quantifies over arbitrary finite source tables and input words. Table 4 locates the principal interfaces; all names are in namespace U20.

TABLE 4. Proof interfaces for the main construction. The supplement gives the full statements, dependencies and re-production commands.

Claim	Lean interface
Ordinary queue call	<code>RawProgram.execute</code> , <code>RawProgram.bounded</code>
Two source passes	<code>Guarded.source_cycle</code> ; <code>DirectPhase.forward</code> , <code>DirectPhase.returning</code>
Copier and epoch	<code>RefreshCopier.complete</code> ; <code>RefreshAutonomous.epoch</code>
Constructive compiler	<code>DirectAdapter.unitCertified</code> ; <code>DirectStop.compile</code>
Theorem 1.1	<code>Finalized.finite_input_universality_and_readback</code>
Resources and readback	<code>Finalized.reference_polynomial_physical</code>
Theorem 7.1, source-only space	<code>PaperInterfaces.reference_polynomial_space_</code> <code>contract</code>
Reader cost	<code>PaperInterfaces.halted_reader_cost</code>

The resource and readback proofs initially produce separate halting witnesses. Determinism identifies their terminal times and configurations, giving one execution with all the stated guarantees. The paper-facing space contract also combines the initial-size bound with the endpoint estimate to obtain the coefficient S_T depending only on the source table.

Eight public paper-facing contracts and two private helpers are checked against the companion. Their transitive dependencies use only `propext`, `Classical.choice` and `Quot.sound`; the audit rejects unfinished proofs and additional axioms. The complete library is supplied as accompanying software [19], with the paper-facing checks beside the manuscript.

The executable checks have narrower, complementary roles. Kernel reduction verifies the 2,999-transition fixture. Independent simulators check the hardware, complete small calls and installations, and replay the two exported reference examples through readback. The quantified universality proof is independent of those finite tests.

11. RELATED WORK

11.1. Instruction count and simulation time. The preceding U_{21} preprint and the older bi-tag route on U_{20} have conservative sixth-degree time and cubic-space bounds when the input length is $O(t)$. The direct refresh compiler changes the simulation route; the instruction saving alone does not improve the exponent. Table 5 compares selected routes.

TABLE 5. Selected ordinary finite-input constructions.
Source tables are fixed and represented input length is $O(t)$.
Bounds concern the specified simulation routes, not optimal runtimes of the hardware.

Construction	States $\times$ symbols	Defined rules	Time bound
Rogozhin / 2-tag [17, 13]	4×6	22	$O(t^4 \log^2 t)$
Neary–Woods bi-tag [14]	5×5	22	$O(t^6)$
Preceding U_{21} [18]	5×5	21	$O(t^6)$
U_{20} , direct reference	5×5	20	$O(t^3)$
Neary–Woods direct [12]	8×4	30	$O(t^2)$

These are asymptotic upper bounds. The U_{20} reference compiler has astronomically large fixed-source constants; this table establishes no practical speed ordering or measured crossover between routes.

The 30-rule entry is counted directly from Definition 10 in [12, Section 5.4]: its 8×4 table leaves precisely (u_2, λ) and (u_2, δ) undefined, hence has $32 - 2 = 30$ defined instructions. Halting pairs are not counted as instructions.

The Rogozhin row uses a later efficient 2-tag simulation. Woods and Neary’s 2006 bounds were $O(t^4 \log^2 t)$ for 2-tag systems and $O(t^8 \log^4 t)$ for the resulting small universal machines [20]; Neary’s 2008 thesis improves them to $O(t^2 \log t)$ and $O(t^4 \log^2 t)$, respectively [13, Section 5.3]. The 30-rule direct construction uses one-way tapes and a normalized source model. These are asymptotic comparisons, not a common timing experiment. Weak machines permit periodic infinite backgrounds on both sides and semi-weak machines on one side [15, 21, 16].

The small-machine literature also includes Minsky’s tag-system construction [7], Baiocchi’s 7×4 machine with 25 commands and explicit output

recovery [8], and Kudlek–Rogozhin’s 3×9 machine [9]. State–symbol products in such results must not be read as counts of defined instructions.

Initial tape conventions also matter. Woods–Neary’s 17-rule 4×5 machine is semi-weak, with a periodic infinite background on one side [21]. Watanabe reported a 21-rule 7×3 machine [10], but Neary and Woods state that they could not locate its construction and infer, rather than verify, that it is semi-weak [16, Section 3.1, footnote 3]. Even if Watanabe’s reported construction satisfies the standard model, its 21 rules do not undercut the present 20-instruction upper bound. Margenstern–Pavlotskaya study Turing machines coupled to a finite automaton [11], which changes the machine model. These comparisons therefore require explicit model and encoding conventions.

The self-published U_{21} comparison is an unrefereed predecessor; the present proof is self-contained. A complete priority audit of the Russian-language Rogozhin sources and later constructions remains open.

11.2. Relation to verified universal simulation. Machine-checked universality and quantitative simulation predate this construction. Asperti and Ricciotti’s Matita development distinguishes the 2012 construction, whose termination proof was then in progress, from the completed termination proof in their 2015 treatment; the latter explicitly leaves its efficiency claim unformalized [1, 2]. Xu, Zhang and Urban mechanize Turing machines and computability in Isabelle/HOL [22]; the maintained AFP universal-machine artifact includes a strengthened halting result [23].

Forster, Kunze and Wuttke verify a six-tape interpreter with linear time and constant-factor space overhead for a fixed source machine, and quantitative compilation to single-tape machines [5]. Dalvit and Thiemann verify a multitape-to-single-tape translation with bound $O(f(n)^2 + n)$ [4]. Carneiro formalizes a universal partial recursive function in Lean [3]. These works establish semantic and quantitative precedents. The present contribution combines a fixed twenty-instruction table with constructive finite input, execution bounds and original-source readback.

12. SCOPE AND OPEN PROBLEMS

12.1. Scope of the claims. Twenty counts defined target instructions. Compiler tables, finite initial encodings and the separate output reader can be much larger. The bounds concern target transitions and all retained tape. Host compilation and reader execution have their own cost models. The streaming encoding result fixes the source table; it supplies no uniform polynomial compiler bound as that table varies.

The large completed examples use compressed queue execution and functional decoding of the saved tape. Their calculated transition totals follow exact local formulas; they are not completed step-by-step hardware runs or instruction-by-instruction executions of the five-tape reader. Neither example reaches an autonomous periodic refresh epoch. These boundaries matter when interpreting the experimental evidence.

The supplement describes checked optimizations and a policy that selects smaller initial tapes. Its correctness and size guarantees hold for all candidates; the reference time bound applies to the reference route. A policy minimizing initial size need not minimize runtime. The result is an explicit upper bound, with neither a global minimality claim nor a practical speed record.

12.2. Questions suggested by the construction. Can a standard finite-input machine use 19 instructions? Can twenty instructions be achieved with state–symbol product 24? Corresponding lower bounds would help distinguish limitations of this architecture from limitations of the model.

The immediate compiler question is whether the fixed-source constants can be reduced substantially. The formal template machinery already supports multiple powers; binary powers may shorten the unit-power template, but the resulting banks, inputs and costs must be rebuilt and measured together. A smaller certified compiler would also make an independent execution of a complete autonomous refresh epoch more accessible. Uniform original-source bounds for the optional optimized routes, and sharper constants for the reference route, remain useful targets.

ACKNOWLEDGMENTS

The construction builds on Neary and Woods’ small-machine architecture and the preceding U_{21} development. AI-assisted tools were used in the development and preparation of this paper. The Lean companion and independent execution checks provide the stated evidence; responsibility for the manuscript and its claims remains with the author.

REFERENCES

- [1] A. Asperti and W. Ricciotti, *Formalizing Turing machines*, in WoLLIC 2012, Lecture Notes in Comput. Sci. **7456**, Springer, 2012, 1–25. doi:10.1007/978-3-642-32621-9_1.
- [2] A. Asperti and W. Ricciotti, *A formalization of multi-tape Turing machines*, Theoret. Comput. Sci. **603** (2015), 23–42. doi:10.1016/j.tcs.2015.07.013.
- [3] M. Carneiro, *Formalizing computability theory via partial recursive functions*, in ITP 2019, LIPIcs **141**, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019, 12:1–12:17. doi:10.4230/LIPIcs.ITP.2019.12.
- [4] C. Dalvit and R. Thiemann, *A verified translation of multitape Turing machines into singletape Turing machines*, Archive of Formal Proofs, 30 November 2022. AFP entry and formal artifact.
- [5] Y. Forster, F. Kunze and M. Wuttke, *Verified programming of Turing machines in Coq*, in Certified Programs and Proofs (CPP 2020), ACM, 114–128. doi:10.1145/3372885.3373816.
- [6] L. de Moura and S. Ullrich, *The Lean 4 theorem prover and programming language*, in Automated Deduction—CADE 28, Lecture Notes in Comput. Sci. **12699**, Springer, 2021, 625–635. doi:10.1007/978-3-030-79876-5_37.
- [7] M. L. Minsky, *Size and structure of universal Turing machines using tag systems*, in Recursive Function Theory, Proc. Sympos. Pure Math. **5**, American Mathematical Society, 1962, 229–238. doi:10.1090/pspum/005/0142452.

- [8] C. Baiocchi, *Three small universal Turing machines*, in MCU 2001, Lecture Notes in Comput. Sci. **2055**, Springer, 2001, 1–10. doi:10.1007/3-540-45132-3_1.
- [9] M. Kudlek and Y. Rogozhin, *A universal Turing machine with 3 states and 9 symbols*, in DLT 2001, Lecture Notes in Comput. Sci. **2295**, Springer, 2002, 311–318. doi:10.1007/3-540-46011-X_27.
- [10] S. Watanabe, *4-symbol 5-state universal Turing machine*, Information Processing Society of Japan Magazine **13** (1972), no. 9, 588–592 (Japanese). Publisher repository record. The reported 7×3 machine is discussed with the qualification in [16, footnote 3]; its full table was not verified here.
- [11] M. Margenstern and L. Pavlotskaya, *On the optimal number of instructions for universal Turing machines connected with a finite automaton*, Internat. J. Algebra Comput. **13** (2003), no. 2, 133–202. doi:10.1142/S0218196703001262.
- [12] T. Neary and D. Woods, *Small fast universal Turing machines*, Theoret. Comput. Sci. **362** (2006), nos. 1–3, 171–195. doi:10.1016/j.tcs.2006.06.002.
- [13] T. Neary, *Small universal Turing machines*, Ph.D. thesis, National University of Ireland, Maynooth, 2008. Author-hosted thesis.
- [14] T. Neary and D. Woods, *Four small universal Turing machines*, Fund. Inform. **91** (2009), no. 1, 123–144. doi:10.3233/FI-2009-0036.
- [15] T. Neary and D. Woods, *Small weakly universal Turing machines*, in FCT 2009, Lecture Notes in Comput. Sci. **5699**, Springer, 2009, 262–273. doi:10.1007/978-3-642-03409-1_24.
- [16] T. Neary and D. Woods, *The complexity of small universal Turing machines: a survey*, in SOFSEM 2012, Lecture Notes in Comput. Sci. **7147**, Springer, 2012, 385–405. doi:10.1007/978-3-642-27660-6_32.
- [17] Y. Rogozhin, *Small universal Turing machines*, Theoret. Comput. Sci. **168** (1996), no. 2, 215–240. doi:10.1016/S0304-3975(96)00077-1.
- [18] M. Schroeder, *A universal Turing machine with 21 instructions*, preprint and Lean companion, version 1.0, 25 September 2026. doi:10.5281/zenodo.22957167.
- [19] M. Schroeder, *U20 finalized Lean companion*, supplementary software, frozen release of 29 September 2026, included in the Version 1 submission compendium for this paper, `U20_Lean_Finalized_2026-09-29.zip`; exact checksum and reproduction procedure in the accompanying technical supplement. doi:10.5281/zenodo.23095173.
- [20] D. Woods and T. Neary, *On the time complexity of 2-tag systems and small universal Turing machines*, in FOCS 2006, IEEE, 2006, 439–448. doi:10.1109/FOCS.2006.58.
- [21] D. Woods and T. Neary, *Small semi-weakly universal Turing machines*, Fund. Inform. **91** (2009), no. 1, 179–195. doi:10.3233/FI-2009-0039.
- [22] J. Xu, X. Zhang and C. Urban, *Mechanising Turing machines and computability theory in Isabelle/HOL*, in ITP 2013, Lecture Notes in Comput. Sci. **7998**, Springer, 2013, 147–162. doi:10.1007/978-3-642-39634-2_13.
- [23] J. Xu, X. Zhang, C. Urban, S. J. C. Joosten and F. Regensburger, *Universal Turing Machine*, Archive of Formal Proofs, 2019, revised 2022. AFP entry and formal artifact.

INDEPENDENT RESEARCHER