UNIVERSAL COMPUTATION WITH 144 RESIDUE CLASSES: A ONE-STATE LINEAR OPERATOR ALGORITHM

MICHAEL SCHROEDER

Abstract. We construct an explicit universal one-state linear operator algorithm with 144 residue-selected rules acting on a single nonnegative integer. This reduces the modulus reported by Kaščák (1992) from 396 to 144, a decrease of 252. The construction combines arithmetic invariants that exclude unreachable branches with a quotient-parity test that selects the return operation as division finishes. Every continuing rule gives an exact nonnegative successor on its entire residue progression. We supply the complete numerical table and a Lean 4 proof of universality under explicit computable input and output encodings. The simulation realizes every unary partial recursive function and preserves nontermination. Consequently, its halting set is computably enumerable complete.

1. Introduction

Universal computation can be expressed by a fixed collection of arithmetic rules acting on a single integer. In a one-state linear operator algorithm, that integer stores the entire evolving configuration, and its residue selects the next affine update. The table is fixed; a program and its data are supplied through the initial integer.

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$. An algorithm of modulus m consists of triples (a_r, b_r, c_r), indexed by $0 \le r < m$. At state n, it selects $r = n \bmod m$. A row with $c_r = 0$ is a successful halt; otherwise the next state is the unique natural number n' satisfying

$$c_r n' = a_r n + b_r.$$

The equation is interpreted over the integers. Our construction ensures that the quotient exists and is nonnegative for every continuing raw input. Write $F(n)$ for the eventual halt value, leaving $F(n)$ undefined when execution continues forever.

Date: 28 September 2026.

2020 *Mathematics Subject Classification.* Primary 03D10; Secondary 68Q04, 68V20.

Key words and phrases. one-state linear operator algorithm, generalized Collatz function, small universal machines, undecidability, counter machines, FRACTRAN, Lean 4.

DOI: 10.5281/zenodo.23018009.

Author ORCID: 0009-0004-3249-0195.

We normalize continuing triples by $\gcd(a_r, b_r, c_r) = 1$, with $a_r \geq 0$ and $c_r > 0$, and write halt rows as $(0, 0, 0)$. The coefficient size of a table is $\max_r \max(|a_r|, |b_r|, |c_r|)$.

Theorem 1.1 (An explicit universal arithmetic machine)**.** *The table in Appendix A defines an algorithm of modulus* 144 *with the following properties.*

(i) *Every continuing natural input has an exact nonnegative successor. The successful-halting residues are exactly* 43 *and* 139*.*

(ii) *The coefficient size is* 524261*. Four residues,* $104, 105, 124, 125$*, are filled by identity rules; the simulation occupies the remaining* 140 *residues.*

(iii) *For an effective enumeration* (F_e) *of unary partial recursive functions, there are total computable encodings* $c(e, x)$ *and* $d_0(n)$ *such that*

$$d_0(F(c(e, x))) = F_e(x)$$

as partial functions, including equality of their domains.

(iv) *The successful-halting set* $\mathcal{H} = \{n : F(n) \text{ is defined}\}$ *is computably enumerable complete under many-one reductions, and hence undecidable.*

The quantifier over inputs in the first assertion is unrestricted. The simulation proof uses invariants on specially encoded configurations, whereas the numerical rules remain valid even outside those configurations. The third assertion supplies a fixed machine for all programs, and preserves divergence as well as successful outputs.

1.1. **Context and contribution.** Conway established arithmetic universality by simulating counter programs with rational multiplications selected by divisibility [2]. His later FRACTRAN language includes POLYGAME, a fixed universal program whose initial integer supplies the program index and input [3]. Thus both computation on one integer and a fixed universal arithmetic interpreter have classical precedents. The question pursued here is how compactly such an interpreter can be realized by an explicit residue table, with a proof connecting that table to its universal behavior.

Kaščák reported a universal one-state linear operator algorithm of modulus 396 [7]. The reported moduli satisfy $144/396 = 4/11$: the present construction uses 252 fewer indexed rows, a reduction of approximately 63.6%. Supplement, Section S1, gives his basic definitions and proves their compatibility with our table and decoded partial function.

The register interpreter realizes the 32-instruction reduction described by Gregušová and Korec [6, p. 315]. Their reduction permits input and output coding and uses two simulated data registers; the interpreter itself has seven active registers. Section 4 identifies the edits to their 37-instruction table. This construction is distinct from the strongly universal 32-instruction register machine in Korec's later work [8]. Accordingly, Theorem 1.1 states the encodings explicitly. Ordinary counter-machine universality provides the

classical starting point [11]; the separate Lean proof starts from Mathlib's proved partial-recursive foundation.

The main constructional improvement occurs in a division routine shared by two interpreter operations. On an exact division by seven, the parity of the quotient determines whether the return operation must restore that quotient or double it. Both outcomes can be distinguished modulo 16 at the instant the countdown reaches zero. This removes an additional arithmetic pass. A separate odd-input branch is retained because the shared test alone would give the wrong answer there. Section 5 proves the distinction and identifies a reachable instance of that exception.

The formalization concerns this explicit arithmetic system. Carneiro developed Mathlib's partial-recursive computability theory, including universality and halting undecidability [1]. Larchey-Wendling formalizes reductions through FRACTRAN and two-counter machines in Coq [9]. Our contribution is the concrete 144-rule construction, its global arithmetic validity, and the verified connection from that literal table to arbitrary partial recursive functions.

The new arithmetic work lies in the invariant-based branch allocation, the shared quotient-parity routine, and their realization by the displayed scalar table. Section 6.2 quantifies these contributions.

The proof follows the compilation from source programs to the scalar table. Section 2 gives a worked example and the encodings; Sections 3–6 prove the simulation and its arithmetic validity. Section 7 states the Lean interface and summarizes resource use and reproduction. Appendix A prints all 144 rules. The mathematical supplement contains the complete finite inventories, extended model comparisons, and supplementary proofs and executions.

Scope. The comparison with Kaščák concerns reported moduli: the basic definitions are compatible, but the complete chapter's later input/output conventions have not been checked, and no comparison of speed, coefficients, or input length is made. We claim no smallest-known or globally minimal modulus, shortest description, or unrestricted coefficient optimum. Execution costs use ordinary binary storage under the stated encodings; no polynomial simulation overhead or optimal aggregate bound is claimed, and scalar-step savings do not establish a bit-complexity improvement. Diagnostic runs, solver checks, and certificate counts are cross-checks, not proofs of universality. Lean verifies the explicit table, global arithmetic validity, simulation, computable wrappers, universality, stopped halting set, and the two-cycle family with its persistence in the companion. The conditional coefficient optimum, remaining dynamical results, predecessor comparisons, long execution counts, and resource propositions in the supplement remain outside the formalization. The closed Mathlib theorem supplies a program integer for each partial recursive function and computable fixed wrappers; it does not formalize a uniform computable compiler from Mathlib program codes. The paper's three-counter route uses the explicit effective compiler of Section 3.

2. A worked example and the encodings

2.1. **A computation that can be checked by hand.**

Example 2.1 (Three transitions to a halt)**.** Start with $n_0 = 158$. Four rows of the published table determine the entire execution:

t	n_t	r_t	Selected triple	Next state
0	158	14	$(9, 482, 16)$	119
1	119	119	$(16, 355, 9)$	251
2	251	107	$(2, 749, 9)$	139
3	139	139	$(0, 0, 0)$	HALT

For the first step, $158 \bmod 144 = 14$, so the selected row gives

$$n_1 = \frac{9 \cdot 158 + 482}{16} = \frac{1904}{16} = 119.$$

The next two steps are

$$n_2 = \frac{16 \cdot 119 + 355}{9} = 251, \qquad n_3 = \frac{2 \cdot 251 + 749}{9} = 139.$$

The final residue is 139, a halt row. Thus $F(158) = 139$ after exactly three continuing transitions.

This example uses the literal scalar rules. It shows why the residue must be recomputed after every update: at 251, the relevant row is $251 \bmod 144 = 107$. The halt row records completion at the current integer; it does not divide by zero or take another step. The example is a small raw-input execution. Rows 6 and 67 give the equally short nonhalting example $6 \mapsto 67 \mapsto 6$; Supplement, Proposition S7.1, proves an infinite family of such two-cycles.

2.2. **How one integer represents several counters.**

The basic encoding idea is elementary. The counter triple $(R_0, R_1, R_2) = (2, 1, 0)$ is represented by

$$A = 2^{R_0} 3^{R_1} 5^{R_2} = 12.$$

Incrementing R_1 multiplies A by 3, giving 36, which represents $(2, 2, 0)$. A positive decrement of R_0 divides by 2, giving 18, which represents $(1, 2, 0)$. Divisibility by the selected prime therefore decides whether the corresponding register is positive. The general construction repeatedly uses this correspondence, then implements the arithmetic using a fixed finite control.

Prime packing alone does not provide the final one-state algorithm: its divisions and branch choices still need to be executed. The scalar encoding $E_s(X, Y)$ implements those operations and embeds their control in residues. This second encoding explains both the compact finite table and the large integers appearing in universal computations.

2.3. **The compilation layers.** There are three encodings to keep in view. First, prime powers pack register values into an integer N, so that divisibility tests implement register tests. Second, two explanatory counters X, Y perform the required divisions and transfers. Third, an expression

$$E_s(X,Y) = K_s 2^X B_s^Y + o_s$$

packs those counters and their control s into the final scalar state. The constants K_s, o_s arrange that a residue of $E_s(X,Y)$ identifies both the control and the branch to take. The controls are part of the proof of the encoding; the final algorithm stores only the scalar integer.

The principal quantities have different roles:

Quantity	Meaning
e, C_e	A finite source program and its fixed positive program integer
x, z	Source input and native interpreter input; $z = 3^x$ in the three-counter route
N	Prime encoding of the interpreter's seven registers
s, X, Y	An arithmetic control and its two explanatory counters
n	The sole evolving state of the final algorithm, $n = E_s(X,Y)$

Only at a guarded instruction boundary do we usually have $X = N, Y = 0$. Inside a routine, the appropriate weighted sum of X, Y represents the quantity being divided or transferred.

2.4. **Input and output conventions.** Let e describe an ordinary three-counter program starting at label 1 with registers $(0, x, 0)$, halting at label 0, and returning R_0. The compiler constructs a positive integer C_e, independently of x, and the scalar input is

$$c(e,x) = 9\,2^{\,17 C_e 13^{3^x}} + 40. \tag{2.1}$$

The table is unchanged when the program changes. Only C_e and hence the starting integer change.

For a prime p and a positive integer u, let $v_p(u)$ denote the exponent of p in u. Define the total output decoder by

$$d_0(n) = \begin{cases} v_2(v_{11}(N)), & n \bmod 144 \in \{43, 139\}, \\ & n = 16\,2^N + 107, \quad N \geq 1, \quad v_{11}(N) > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{2.2}$$

The exponent N, when it exists, is unique. Testing the displayed format and extracting its exponent are total finite algorithms. These operations belong to the external decoding convention, not to the scalar rule table. The successful outputs of compiled programs always satisfy the first case; the second case makes d_0 a total function on arbitrary integers.

The three-counter interface above is verified for every finite three-counter program. Section 7 gives the input and output wrappers for the separate Mathlib route.

3. Compiling a universal source language

The compiler has two tasks: express an arbitrary source computation in a language understood by a fixed interpreter, and encode that finite instruction list as a positive integer. The first task uses standard prime packing; the second uses a negative Chinese-remainder convention. Neither task executes the program being compiled.

3.1. **Ordinary counter programs.** We use the standard universality of ordinary three-counter programs [11]. Their registers R_0, R_1, R_2 take values in $\mathbb{N}$; an instruction either increments one register and jumps, or conditionally decrements a positive register and takes a positive branch, taking a separate zero branch if the register is zero. Label 1 starts, label 0 halts, the initial registers are $(0, x, 0)$, and the output is R_0. This is the only external machine-universality theorem needed.

At source instruction boundaries encode these registers by two ordinary counters:

$$S_0 = 0, \qquad S_2 = A = 2^{R_0}3^{R_1}5^{R_2}.$$

The initial value is $A = 3^x$. An increment of R_i multiplies A by $p_i \in \{2, 3, 5\}$: consume S_2, adding p_i units to S_0 per consumption, then transfer S_0 back to S_2.

For a conditional decrement, consume A using p_i cyclic control phases and add one to S_0 on each wrap. At the end the phase is $r = A \bmod p_i$, and $S_0 = \lfloor A/p_i \rfloor$. If $r = 0$, transfer this quotient back to S_2 and choose the positive branch. If $r > 0$, initialize $S_2 = r$ and replace each unit of S_0 by p_i units in S_2, restoring A before choosing the zero branch. All fixed-length addition chains use unit increments. Every loop decreases a nonnegative counter, so each simulated source instruction terminates.

At source halt transfer the entire positive main counter into S_0, leaving $S_2 = 0$. The final value is

$$S_0 = 2^{R_0}3^{R_1}5^{R_2} > 0, \qquad v_2(S_0) = R_0. \tag{3.1}$$

3.2. **The restricted two-counter language.** The interpreter accepts a language, denoted M2, on S_0, S_2 with instructions

$$\begin{aligned} P \to j: &\quad S_0 \leftarrow S_0 + 1,\ S_2 \leftarrow S_2 + 1;\ \text{go to } j, \\ S_k \to j: &\quad \text{if } S_k > 0,\ S_k \leftarrow S_k - 1 \text{ and go to } j; \\ &\quad \text{otherwise go to } j + 1, \qquad k \in \{0, 2\}. \end{aligned}$$

Label 0 halts. The opcodes for S_0, S_2, P are respectively $0, 1, 2$, and the integer code of an instruction with positive destination j is $h = 3j + \text{opcode}$. The same formula permits destination zero, subject to the exclusion $h \neq 0$ below.

Instruction conventions are substantive here. Dudenhefner shows that changing the permitted counter instructions can change decidability, even among models described as two-counter machines [5]. We therefore give the translation into this particular M2 language explicitly.

An increment of just one ordinary counter is P followed by a forced positive decrement of the other. A preserving jump is P, a forced decrement of S_0, and a forced decrement of S_2. To compile an ordinary conditional decrement with arbitrary destinations, reserve adjacent labels $j, j+1$ for two preserving-jump stubs and emit $S_k \to j$. Fresh labels make adjacency literal, independently of the source destinations.

All jumps to halt pass through a preserving-jump wrapper ending in $S_2 \to 0$, of code 1. The forced decrement is positive, so it reaches label 0. No instruction is $S_0 \to 0$, whose code would be zero. Normalize the positive labels to a contiguous list $1, \dots, v$ and add a final unreachable $P \to$ itself padding instruction if needed, so even unused syntactic zero branches have defined destinations. These are finite, effective graph transformations. Each macro terminates and preserves the intended counter effect, including the halt wrapper. Hence they preserve nonhalting as well as successful outputs.

Lemma 3.1 (restricted-source universality)**.** *The resulting M2 program, on input* $(S_0, S_2) = (0, 3^x)$*, halts exactly when the original three-counter program halts; on halting it has the output convention* (3.1)*. All its codes satisfy* $1 \le h_i \le 3v + 2$*.*

3.3. **Negative Chinese-remainder encoding.** The decoder used below fetches nonzero values of $(-C) \bmod d$, in increasing trial divisor d. The minus sign corrects a sign inconsistency in the printed source. Gregušová and Korec's congruence (3) has $y \equiv 4j + |X| \pmod{p_{r+w}}$ [6, p. 311], whereas their decoder on p. 312 returns the complement $d - (y \bmod d)$ when the remainder is nonzero. To recover $h = 4j + |X|$, that congruence must be $y \equiv -h$; the same correction applies to the unused-instruction codes in their congruence (4). For example, a remainder 1 modulo 7 decodes as 6, while a remainder 6 decodes as 1. The countdown calculation in Lemma 4.2 proves the complement identity. Our two-register reduction replaces $4j + |X|$ by $3j + k$ and retains this corrected negative sign.

Choose consecutive primes $p_r, p_{r+1}, \dots, p_{r+v}$ such that

$$p_r > 3v + 2, \qquad P := p_{r+v} < 2p_r. \tag{3.2}$$

Searching for such a block terminates. Indeed, if beyond some point every block of this fixed length failed the second inequality, each of the finitely many interlaced subsequences of primes would grow at least geometrically. The sum of reciprocals of all primes would then converge, contradicting its standard divergence.

For each prime $p \le p_r$, let p^k be its greatest power strictly below P, and impose $C \equiv 0 \pmod{p^k}$. Impose in addition

$$C \equiv -h_i \pmod{p_{r+i}}, \qquad 1 \le i \le v. \tag{3.3}$$

These moduli are pairwise coprime, so a positive solution C is computable by the Chinese remainder theorem. Every composite $d < P$ divides C: a prime factor above p_r would imply $d \geq 2p_{r+1} > P$, while all prime powers required for factors at most p_r were included. Every prime at most p_r also divides C. At an instruction prime the complement is exactly h_i, because $0 < h_i < p_{r+i}$.

Lemma 3.2 (effective program encoding)**.** *The first v nonzero complements $(-C) \bmod d$, for $d = 1, 2, \ldots$, are precisely $h_1, \ldots, h_v$, in that order. Thus (3.2)–(3.3) define a total effective program encoder $e \mapsto C_e > 0$. The encoder does not execute the source program or decide its halting.*

4. A fixed interpreter for encoded programs

The interpreter separates execution from instruction fetching. Its working block performs one decoded operation. Its decoder recovers the next instruction from the fixed program integer and restores that integer before returning. This separation makes the proof of repeated interpretation a direct induction on source steps.

4.1. **Complete instruction specification.** The interpreter has registers $S_0, S_1, S_2, S_4, S_5, S_6, S_7$; S_3 is unused and fixed at zero. Instructions are $I(k, t)$, which increments S_k and goes to t; $D(k, t)$, which replaces S_k by $\max(S_k - 1, 0)$ and goes to t; and $T(k, t, u)$, which tests $S_k > 0$ without changing it and chooses t or u. Label 0 halts. Supplement, Section S2 gives all 32 instructions, retaining their original, nonconsecutive labels.

For precise provenance, start from the 37-instruction lists in [6, pp. 312, 314], delete labels $7, 8, 14, 15, 18$, and replace only labels $5, 6, 17$ by $T(5, 6, 16)$, $D(5, 9)$, and $I(2, 20)$, respectively. Every other instruction is retained. The list in Supplement, Section S2 and the following proofs specify the resulting machine independently of the historical description.

The native initial configuration is label 1 with

$$S_1 = C > 0, \quad S_2 = z \geq 0, \quad S_0 = S_4 = S_5 = S_6 = S_7 = 0. \tag{4.1}$$

Lemma 4.1 (working block)**.** *Entered at label 1 with $S_5 = 3j + k$, $k \in \{0, 1, 2\}$, and $S_7 = 0$, the interpreter reaches label 20 with $S_5 = 0$, the data-register effect of the coded M2 instruction, and its next instruction index in S_7.*

Proof. Each complete traversal of tests $1, 3, 5$ consumes three units through decrements $2, 4, 6$, and increments S_7 at label 9. There are exactly j traversals. Remainder zero selects the conditional decrement of S_0; remainder one consumes its remaining unit and selects the conditional decrement of S_2; remainder two consumes both units and increments both data registers at $16, 17$. A failed data decrement increments S_7 at 19, implementing destination $j + 1$. All these paths are finite.

□

At initialization the zero value of S_5 acts as a failed $S_0 \to 0$ instruction: it sets the first fetch index to 1 without changing S_2. This bootstrap is not an encoded program instruction.

Lemma 4.2 (decoder)**.** *Entered at label* 23 *with* $S_1 = C > 0$*,* $S_4 = S_5 = S_6 = 0$*, and* $S_7 = i > 0$*, the decoder returns to label* 1 *with* S_5 *equal to the* i*-th nonzero complement* $(-C) \bmod d$*. It restores* $S_1 = C$*, clears* S_4, S_7*, and leaves finite scratch data in* S_6*.*

Proof. Before the trial for d, the countdown registers have sum $S_5 + S_6 = d-1$. Labels 23–25 transfer all C units from S_1 to S_4. Label 26 and the loop 27–28 set $S_5 = d, S_6 = 0$. Each traversal of 29–32 transfers one unit from S_4 back to S_1, and one unit from S_5 to S_6. When the countdown expires before S_4, 27–28 reload it. After exactly C consumptions,

$$S_1 = C, \quad S_4 = 0, \quad S_5 = (-C) \bmod d, \quad S_5 + S_6 = d.$$

Tests 33–35 select reloading, the next trial, or index decrement. A zero complement does not decrement S_7; a positive complement does so at 36. Test 37 returns to the working block precisely when the requested index has been exhausted. Each trial is finite. For every $d > C$ the complement is positive, so every finite index is eventually found.

□

After a working instruction, label 20 halts exactly if the next index is zero. Otherwise 21–22 clear S_6 before another fetch; the working block already cleared S_5. The decoder preconditions therefore recur. Lemmas 3.1–4.2 prove universality of this *fixed* interpreter, including preservation of non-halting.

4.2. **Prime encoding and guarded paths.** Encode its registers in one positive integer

$$N = 11^{S_0}17^{S_1}13^{S_2}5^{S_4}2^{S_5}7^{S_6}3^{S_7}. \tag{4.2}$$

Its native initial value is $N = 17^C 13^z$. A prime divisibility test is exactly a positive-register test. Compress finite paths into the following 16 guarded nodes. A row $(q; p, a, b, t, u)$ means: if $p \mid N$, send N to aN/p and go to t; otherwise send N to bN and go to u.

q	p	a	b	t	u
1	2	1	1	3	10
3	2	1	1	5	12
5	2	3	143	1	20
10	11	1	3	20	20
12	13	1	3	20	20
20	3	3	1	21	0
21	7	1	1	21	23
23	17	5	2	23	27
27	7	2	1	27	29
29	2	7	7	31	31
31	5	17	17	33	33
33	5	5	1	34	35
34	2	7	1	31	27
35	2	2	1	36	23
36	3	1	1	37	37
37	3	3	1	23	1

The positive/zero source paths for these rows are listed in Supplement, Section S2. Every path has at most three primitive instructions. Along a positive branch the tested register is at least one; along a zero branch it is zero. The listed paths resolve every truncated decrement under these assumptions. Adding their register changes and applying unique prime factorization proves all 32 multiplicative identities, without any bound on register values.

4.3. **An inductive source certificate.** A mask $m = \sum_{k=0}^{7} 2^k[S_k > 0]$ records only positivity. The following abstract semantics overapproximates every concrete transition: an increment sets its bit; a test chooses according to its bit; a positive decrement permits either clearing or retaining its bit; a decrement of zero retains zero. Supplement, Section S3 gives the allowed mask set at every source label, containing 431 label/mask pairs.

The initial masks 2 and 6 are included. For each listed pair, every abstract successor is listed at the target label, unless the target is the halt label. This is a finite closure certificate: checking each instruction and its at most two successors proves by induction on execution length that all reachable masks are listed. In particular,

$$\begin{aligned} &q = 29 : S_5 > 0, \quad q = 31 : S_4 > 0, \quad q = 36 : S_7 > 0, \\ &q = 20 : S_5 = 0, \quad q = 37 : S_5 > 0, \quad q = 10,\ S_0 > 0 :\ S_7 > 0. \end{aligned} \tag{4.3}$$

At $10, 12, 21$ the encoded N is odd. At $21, 23, 27$ it is divisible by 3. Odd entries at 27 have $S_6 > 0$, hence $7 \mid N$. These restrictions justify the branch omissions and shared operations below. Induction on execution length extends the invariant to every reachable source configuration.

5. Arithmetic routines and quotient-parity selection

Introduce explanatory counters $X, Y \in \mathbb{N}$. At boundaries representing guarded node q, we normally have $X = N, Y = 0$. The controls below implement the guarded arithmetic by finite counter routines. A notation such as $D10{:}4$ denotes phase 4 of division control $D10$, not a source instruction label. Changes and exits listed here are the full control specification; no extra work register is implicit.

Every control uses work base 9 in the scalar encoding of the next section, except MP and $R9$, which use base 3. A base change occurs only on the exit $R1 \to MP$ or $R9 \to R3$, with $Y = 0$ on both sides.

5.1. **Exact divisions.** For divisor p, choose increasing phases $0 = \rho_0 < \cdots < \rho_{s-1} < p$. Let the stride at phase ρ be the distance to the next phase, or $p - \rho$ at wrap. A positive step subtracts this stride from X, advances phase, and adds one to Y on wrap. The invariant is

$$N = X + pY + \rho. \tag{5.1}$$

At phase zero, $X = 0$ exits to the listed transfer; at other phases zero is excluded by exact divisibility. The complete exact-division list is

Division	p	Phases	Exit from phase zero
$D1$	2	0	$R0$
$D3$	2	0	$R1$
$D5$	2	0	$R5$
$D31$	5	$0, 4$	C_0
$D34$	2	0	$R6$
$D36$	3	0	$R2$
$D27$	7	$0, 4$	$R13$

If $p \mid N$, at phase zero a positive X is at least p; at phase $\rho > 0$, $X \equiv p - \rho \pmod{p}$ and is at least $p - \rho$. Every decrement is therefore safe. Each positive step decreases X, and the zero exit gives $(X, Y) = (0, N/p)$.

5.2. **Divisions with restoration.** For the following three divisions, apply the same positive phase steps whenever X is at least the stride h. At a low exit $X = j < h$, replace X by $j + \rho$ and keep Y. This gives the exact remainder and quotient. Choose the true transfer if $j + \rho = 0$, and the false transfer otherwise.

Division	p	Phases	True transfer	False transfer	Entry restriction
$D10$	11	$0, 4, 8$	$R2$	$R10$	N odd
$D12$	13	$0, 4, 8, 12$	$R3$	$R12$	N odd
$D23$	17	$0, 3, 6, 9, 12, 15$	$R8$	$R11$	none

The invariant (5.1) proves both branches: a true transfer acts on the quotient; a false transfer reconstructs the original value by adding pY to its remainder. Again, every positive step decreases X.

5.3. **Shared division by seven and quotient parity.** The shared $D21$ routine has phases $0, 4$ and invariant $N = X + 7Y + \rho$.

- At phase 0, if $X \geq 4$, subtract four and enter phase 4. If $X = 0$, enter $R13$ when Y is even, and $R2$ when Y is odd. If $X \in \{1, 2, 3\}$, enter $R7$. All exits in this phase leave the counters unchanged.
- At phase 4, if $X \geq 3$, replace (X, Y) by $(X - 3, Y + 1)$ and enter phase 0. If $X \in \{0, 1, 2\}$, add four to X and enter $R7$.

Write $N = 7Q + R$, $0 \leq R < 7$. Every exit has quotient $Y = Q$ and remainder $X = R$. On an exact exit, parity of Q equals parity of N. Odd inputs represent source node 21, whose positive branch is $N/7$; $R2$ transfers Q unchanged. Even inputs represent node 27, whose positive branch is $2N/7$; $R13$ transfers $2Q$ directly. A nonzero remainder goes through $R7$, restoring N and choosing the appropriate successor by parity.

This sharing has a necessary exception. An *odd* entry at source node 27 requires $2N/7$, whereas odd shared-$D21$ inputs implement $N/7$. Such entries use the separate exact $D27$ routine and $R13$. The exception is reachable: the native configuration $C = 2, z = 0$ reaches source 27 with registers $(0, 1, 0, 0, 1, 0, 1, 1)$, in order $S_0, \ldots, S_7$, and hence $N = 1785$. The required result is 510, not 255.

The reason quotient parity costs no extra control is the identity, for $P = 2^X 9^Y$,

(5.2)

case	$X \geq 4$	$\overset{X=0}{Y \text{ even}}$	$\overset{X=0}{Y \text{ odd}}$	$X = 1$	$X = 2$	$X = 3$
$P \bmod 16$	0	1	9	2	4	8.

Indeed $9^Y \equiv 1 + 8Y \pmod{16}$, and multiplying by 2^X erases the parity term as soon as $X \geq 1$. Phase zero therefore needs six residues modulo 16; phase four needs four residues modulo 8. The split $4 + 3$ uses ten residues in two controls. A three-phase $3 + 3 + 1$ division would also use ten residues, but would need a separate transfer to discover quotient parity after restoration.

Example 5.1 (The residue selects the return operation)**.** Consider two contract-admissible calls to $D21$: $N = 210$ and $N = 357$. Each pair of positive steps subtracts seven from X and adds one to Y, so their exact exits have $(X, Y) = (0, 30)$ and $(0, 51)$, respectively. At $D21{:}0$ the scalar location is $n = 9\,2^X 9^Y + 36$. Thus

Input N	Quotient Y	$9^Y \bmod 16$	$n \bmod 144$	Transfer	Restored X
210	30	1	45	$R13$	60
357	51	9	117	$R2$	51

The two selected rows act by

$$\frac{16n - 507}{3} = 48\,9^{30} + 23, \qquad \frac{16n - 324}{9} = 16\,9^{51} + 28,$$

respectively. These are exactly the entry encodings of $R13$ and $R2$. The first transfer adds two per remaining unit of Y; the second adds one. Consequently the two calls return $2(210/7) = 60$ and $357/7 = 51$. The completion residue has already selected the required multiplier before either transfer begins.

5.4. **Multiplication, transfer, and fused tests.** The three multiplication controls act while $X > 0$ by $(X, Y) \mapsto (X - 1, Y + b)$ and, at $X = 0$, enter a transfer without changing the counters:

Control	b	Zero exit	Work base
MP	11	$R9$	3
$M3$	3	$R2$	9
$M2$	2	$R3$	9

An ordinary transfer of increment d acts while $Y > 0$ by $(X, Y) \mapsto (X + d, Y - 1)$. Its zero exit leaves the counters unchanged and chooses according to the parity of X:

Control	d	Even exit	Odd exit
$R0$	1	$D3{:}0$	$D12{:}0$
$R1$	1	$D5{:}0$	MP
$R5$	3	$D1{:}0$	$D10{:}0$
$R6$	7	$D31{:}0$	$D31{:}0$
$R7$	7	$D34{:}0$	$D23{:}0$
$R8$	5	$D23{:}0$	$D23{:}0$
$R9$	13	excluded	$R3$
$R10$	11	excluded	$M3$
$R11$	17	$M2$	$M2$
$R12$	13	excluded	$M3$
$R13$	2	$D21{:}0$	excluded

The excluded parities are ruled out by the contracts below. Multiplication and transfer compose to give, for example, $MP/R9 : N \mapsto 143N$, while $R10/M3/R2$ restores a failed division by 11 and multiplies by 3.

The fused transfers $R2, R3$ both increment X by one while decrementing positive Y. At $Y = 0$ their exits are

(5.3)

X mod 6	0	1	2	3	4	5
$R2$	$D23{:}0$	–	$D1{:}0$	$D21{:}0$	$D1{:}0$	–
$R3$	$D21{:}0$	HALT	–	$D21{:}0$	–	HALT.

A dash is an excluded case. A HALT entry halts in place; it is not an extra transition. These transfers combine restoration with a divisibility-by-three test and, where needed, a parity test.

Finally, $C_0, \ldots, C_4$ form a transfer ring. While $Y > 0$, add 17 to X, decrement Y, and advance i modulo 5. At $Y = 0$, C_0 exits to $D34{:}0$ on even X and $D27{:}0$ on odd X; every other C_i exits to $D36{:}0$ on even X and $D23{:}0$ on odd X. Entered from $D31$ at $(C_0, 0, Q)$, this ring ends with $X = 17Q$ and $i = Q \bmod 5$. Its phase therefore performs the source test $5 \mid N$ without a separate division routine.

There are 24 division-phase controls, three multipliers, eleven ordinary transfers, two fused transfers, and five ring controls: 45 in total. Every division or multiplication body decreases X; every transfer body decreases Y. Each call to these routines terminates.

5.5. **Boundary correspondence and unbounded contracts.** The following dispatch convention identifies guarded boundaries with counter boundaries $(X, Y) = (N, 0)$. An even/odd split refers to N; recursively indicated dispatches change no counter.

Guarded node	Counter dispatch
1	even: $D1{:}0$; odd: $D10{:}0$
3	even: $D3{:}0$; odd: $D12{:}0$
5	even: $D5{:}0$; odd: MP
$10, 12, 21, 23$	corresponding $Dq{:}0$
20	$R3$
27	even: $D21{:}0$; odd: $D27{:}0$
29	$D34{:}0$
31	$D31{:}0$
33	dispatch 34 if $5 \mid N$, otherwise 35
34	even: $D34{:}0$; odd: $D27{:}0$
35	even: $D36{:}0$; odd: $D23{:}0$
36	$D36{:}0$
37	$R2$
0	HALT

The source masks in Supplement, Section S3 justify this dispatch. For instance, node 29 always has $2 \mid N$, so $D34/R6$ implements its $7N/2$ branch.

Node 31 always has $5 \mid N$; $D31$ and the C ring implement $17N/5$ and the following tests. Node 36 always has $3 \mid N$. Node 20 has odd N, so its divisibility-by-three decision is represented by the odd exits of $R3$. Node 37 has even N, so $R2$ combines its test with the next parity dispatch. The phase ring accounts for the potentially omitted node 33 on return from 31.

The proof uses arithmetic contracts at each control. In a division phase they have the form $W = X + pY + \rho$, together with divisibility and parity conditions inherited from the source invariant. In a transfer they use the corresponding conserved weighted sum. Supplement, Section S5 lists every contract. Their role is to justify the omitted branches and the dispatch between routines for unbounded counter values.

Lemma 5.2 (counter invariant and simulation)**.** *Native initialization $X = 17^C 13^z, Y = 0$, with $C > 0$, at $D10{:}0$ satisfies these contracts. Every specified continuing counter step preserves the target contract. The omitted branches are unreachable from configurations satisfying the contracts. At source boundaries satisfying the source-mask invariant, the routines and dispatch above simulate the guarded interpreter, with the stated halting convention.*

Proof. The exhaustive cases are proved in Lean under `Oloa/Counter/`. In namespace `Oloa.Counter`, theorem `release_contract_iff` binds the release predicates to Supplement, Section S5. Theorem `native_valid` proves initialization for $C > 0$. Theorems `progress` and `contract_preserved` prove permitted progress and contract preservation for arbitrary natural counters. Theorem `guarded_simulation` covers every guarded branch under the source invariant; `stutter_rank` handles omitted tests. Their scalar composition, `native_halts_iff`, proves both forward simulation and halting reflection. These theorems supply the exhaustive proof; the following calculations explain it.

Initially X is positive, odd, divisible by 17, and coprime to 11, giving the $D10$ contract. Every positive division step preserves (5.1); every transfer preserves its displayed W. For the ring, a positive step changes X by 17 and phase by one, preserving the extra congruence. Multiplication preserves its weighted form. Thus all positive-body implications are direct affine identities and congruences.

At a division exit write $W = pQ + R$. The exact routines have $R = 0$; the restoration routines have $0 \le R < p$. Coprimality with the fixed factors in the table gives the target divisibilities. As representative cases, the true $D10$ exit has W divisible by $11 \cdot 17 \cdot 3$, with odd $Q = W/11$, so it enters $R2$ in class 3 modulo 6. A false $D10$ exit restores odd W divisible by 17 and then multiplies it by 3, again giving that $R2$ class. A true $D12$ exit has odd quotient divisible by 17, which is an admissible $R3$ value; a false exit restores the input before the same multiplication by 3.

For $D21$, if $R = 0$ and Q is odd, Q is an odd multiple of 51, hence $R2$ exits to $D21$. If Q is even, Q is an even multiple of 15, and $R13$ restores $2Q$,

a multiple of 30. With $R > 0$, $R7$ restores W: an odd result satisfies $D23$ through its factor 17; an even result satisfies $D34$ through its factor 30.

For $D23$, a true exit through $R8$ has weighted value $5W/17$, divisible by 15, and its completed value satisfies $D23$ through factor 5. A false exit can only occur with $5 \mid W$; since $3 \mid W$, restoration through $R11$ gives a multiple of 15 and doubling gives the $M2/R3$ multiple of 30.

The C ring enters with $Q = W_{D31}/5$, divisible by 21, so its weighted value $17Q$ is divisible by 357. At a zero exit, $X = 17Q$ and $i \equiv Q \pmod 5$. At $i = 0$, X is divisible by 1785; parity therefore selects a valid $D34$ or exact odd $D27$ call. At $i \neq 0$, parity selects $D36$ or $D23$ with their required factors. The ordinary transfer exits then follow immediately from their preserved values and (5.3).

The external certificate provides an independent check of the exit implications by periodic reduction. Fix the small counter in a branch and call the remaining counter z. Take L to be the least common multiple of $2, 6$ and every source/target congruence modulus. For each $0 \leq r < L$, choose the least $z \equiv r \pmod L$ satisfying source positivity. If its source contract holds, check the indicated target, counter nonnegativity, and target positivity. Congruences then recur for $z + kL$, and target positivity persists because its slope is nonnegative. The supplied certificate performs 76 such fixed-counter checks, examining 54,108 classes, of which 1,687 are admissible infinite progressions. Each checked representative therefore covers its entire admissible progression.

The effects of each terminating routine are the quotient, restoration, or multiplication already proved. Substituting these effects into the guarded table and the dispatch table establishes simulation. Only pure tests are removed without a counter step. Their dependency graph is acyclic, and at most two such tests can occur consecutively; assigning each omitted test the longest remaining path length gives a strictly decreasing rank. Every nontrivial guarded computation therefore has a finite, positive counter implementation, apart from these bounded test omissions.

□

6. Encoding the controls in residues

The last compilation step removes the explanatory controls. Each control receives a scale and an offset. The chosen residue classes are disjoint, so a single remainder computation recovers the current control and the arithmetic branch. Affine identities then turn every counter update into one scalar rule.

6.1. **Residue tests and locations.** Encode a counter configuration at control s by

$$E_s(X, Y) = K_s 2^X B_s^Y + o_s, \qquad B_s = \begin{cases} 3, & s = MP, R9, \\ 9, & \text{otherwise.} \end{cases} \tag{6.1}$$

Supplement, Section S4 gives all K_s, o_s and allocated residues. The following elementary residue observations explain the allocation.

- Modulo 2, $2^X B^Y$ distinguishes $X = 0$ from $X > 0$, for either odd base.
- Modulo 3, it is zero for $Y > 0$ and is 1 or 2 for $Y = 0$ according as X is even or odd.
- For a stride $h \leq 3$ in base 9, modulo 2^h the high branch $X \geq h$ gives zero, and each low $X = j < h$ gives 2^j, since $9 \equiv 1 \pmod{2^h}$.
- For an odd division invariant $N = X + pY + \rho$ with odd p, $Y \equiv 1 - \rho - X \pmod 2$. At low $X = j < h \leq 4$ the residue is $2^j 9^{(1-\rho-j) \bmod 2} \bmod 2^h$. Its two-adic valuation distinguishes j. This permits four-unit chunks in $D10, D12$.
- Equation (5.2) supplies the shared quotient-parity test.
- Modulo 9, a positive Y gives zero. At $Y = 0$, the powers 2^X are $1, 2, 4, 8, 7, 5$ for $X \bmod 6 = 0, 1, 2, 3, 4, 5$, implementing (5.3).

For a residue test modulo $d \mid 144$, take $K_s = 144/d$. Then the observation $h \bmod d$ selects residue $o_s + K_s h \bmod 144$. The actual scales are $9, 16, 18, 36, 48, 72$. The allocated sets in Supplement, Section S4 are pairwise disjoint and their union has size 140. Every location satisfies $K_s + o_s \geq 1$, so all encoded integers are positive, even where the offset is negative.

6.2. **Residue budget and the effect of the invariants.** The following budget is regenerated from the scalar locations in the frozen release. "Before pruning" counts the possible residues of each $E_s(X, Y)$ for unrestricted $X, Y \geq 0$ at the final scales, then sums these local demands over the indicated controls. The unpruned sets may overlap; the allocated sets are disjoint.

Control family	Controls	Before pruning	Allocated
Exact divisions	9	18	16
$D10$	3	16	14
$D12$	4	20	17
$D21$	2	10	10
$D23$	6	23	23
Multipliers	3	6	6
Ordinary transfers	11	33	29
Fused transfers $R2, R3$	2	14	10
Five-control ring	5	15	15
Total	45	155	140

These counts are exhaustive. For the actual test moduli $d = 144/K_s$, the values $0 \leq X < 10$ represent every power-of-two residue: powers vanish above the two-adic threshold when d is a power of two, and have period at most six when $d = 3$ or 9. For base 9, the values $Y = 0, 1, 2$ cover its

residues modulo each such d. The base-3 controls only use $d = 2, 3$, where the same representatives suffice.

The contracts remove fifteen local demands. Exact division excludes one low exit in each of $D27{:}4, D31{:}4$. Odd-input invariants remove one zero-count parity case in each of $D10{:}0, D10{:}4, D12{:}0, D12{:}4, D12{:}8$. The fixed return parities of $R9, R10, R12, R13$ remove four more, and the fused dispatchers $R2, R3$ each exclude two zero-work exits. Thus the reductions are $2 + 5 + 4 + 4 = 15$, giving $155 - 15 = 140$ occupied residues, including the two halts. The four remaining positions receive identity rules.

The archived predecessor and the exact scalar-step comparison are documented in Supplement, Section S6.

6.3. **Affine compilation.** For a continuing counter effect $(s, X, Y) \mapsto (t, X + \delta_x, Y + \delta_y)$ with the same work base, write in lowest terms

$$\alpha = \frac{K_t}{K_s} 2^{\delta_x} B_s^{\delta_y} = \frac{a}{c}, \qquad b = co_t - ao_s. \tag{6.2}$$

Then

$$\frac{aE_s(X, Y) + b}{c} = E_t(X + \delta_x, Y + \delta_y). \tag{6.3}$$

The two changes of work base occur at $Y = 0$, with $\delta_y = 0$, so (6.3) holds there as well. Formula (6.2), applied to the preceding control specification and Supplement, Section S4, generates all 138 occupied continuing rows in Appendix A. The remaining occupied cases are HALT at residues $43, 139$ of $R3$. The four unallocated residues get $(1, 0, 1)$.

Disjointness ensures that a legal encoding cannot select another control's rule or a spare rule. Branch coverage follows from the residue observations and Lemma 5.2. Since $\gcd(a, c) = 1$, the triple in (6.2) is already normalized. The identities can be checked row by row by comparing slope and intercept, independently of execution examples.

Lemma 6.1 (validity on all raw inputs)**.** *Every continuing row in Appendix A maps its entire residue progression to* $\mathbb{N}$.

Proof. For each of the 142 continuing rows, direct integer calculation gives

$$a_r \geq 0, \quad c_r > 0, \quad c_r \mid 144a_r, \quad c_r \mid (a_r r + b_r), \quad a_r r + b_r \geq 0. \tag{6.4}$$

For $n = r + 144q$, $q \in \mathbb{N}$, the successor is

$$\frac{a_r r + b_r}{c_r} + \frac{144a_r}{c_r} q \in \mathbb{N}.$$

All entries needed for these finite calculations are printed in Appendix A. For example the largest negative intercept is at residue 4:

$$\frac{131072n - 524261}{9}, \qquad n = 4 + 144q,$$

whose value is $3 + 2097152q$. Thus a negative coefficient does not entail a negative raw successor.

□

6.4. **Composition and preservation of nonhalting.**

Proof of Theorem 1.1. Given a source program e, use Lemmas 3.1 and 3.2 to compute C_e. Use the native input $z = 3^x$. Its prime encoding is $N = 17^{C_e}13^{3^x}$; the initial guarded tests at 1 are zero branches, so the first counter control is $D10{:}0$ with $X = N, Y = 0$. Its location is $K = 9, o = 40$, giving (2.1).

Lemmas 4.1 and 4.2 prove correct interpretation of every compiled instruction. The path identities and source certificate give the guarded simulation. Lemma 5.2 gives finite correct counter implementations, and (6.3) gives each corresponding scalar step. Halting occurs at $R3$ with $Y = 0$, whose location has $K = 16, o = 107$. Its scalar output is therefore $16\,2^{N_{\text{final}}} + 107$. At a compiled successful halt, $v_{11}(N_{\text{final}}) = S_0 > 0$, and (3.1) gives the decoded result (2.2).

A finite halting source computation yields a finite scalar computation. Conversely, legal encodings remain in their allocated domains; successful halt can occur only at the specified interpreter halt. Each counter routine is terminating, each decoder fetch is terminating, and the omitted pure tests have no infinite path. An infinite compiled computation therefore yields infinitely many scalar transitions and cannot acquire an accidental successful halt or become trapped inside a finite-call routine. This proves equality of partial functions. Lemma 6.1 supplies validity on every raw input. Inspection of the normalized triples gives coefficient measure 524261. Finally, repeated exact iteration semidecides $\mathcal{H}$. For any computably enumerable set, choose a partial recursive function with that domain and apply the total input encoding already constructed. This gives a computable many-one reduction to $\mathcal{H}$, proving its completeness and undecidability.

□

7. Formalization, resources, and reproducibility

7.1. **The theorem proved in Lean.** The companion development uses Lean 4 [4] and Mathlib [10], building on Carneiro's computability foundations [1]. It binds a literal Lean table to the canonical CSV and connects the stopped computation to arbitrary partial recursive functions. The principal declarations are

```
Oloa.universal_partial_function
Oloa.halting_undecidable
Oloa.halting_complete
```

For clarity, the closed Mathlib route uses the following wrappers. Define

$$u(0) = 2, \qquad u(x) = 5u(\lfloor x/2 \rfloor) + 3 + (x \bmod 2) \quad (x > 0),$$

and

$$I(C, x) = 9\,2^{17^C 13^{2^{u(x)}}} + 40. \tag{7.1}$$

Here $u(x)$ packs the binary digits of x as digits $3, 4$ in base five, with terminator 2. Let $b(0) = 0$ and, for $s > 0$, set

$$b(s) = \begin{cases} 2b(\lfloor s/5 \rfloor), & s \bmod 5 = 3, \\ 2b(\lfloor s/5 \rfloor) + 1, & s \bmod 5 = 4, \\ 0, & \text{otherwise.} \end{cases}$$

The output wrapper is $D(n) = b(d_0(n))$. The closed statement is

$$\forall g \in \text{Partrec}\ \exists C > 0\ \forall x, \qquad D(F(I(C, x))) = g(x) \tag{7.2}$$

as partial functions. Lean proves that F is partial recursive and that I and D are total computable functions.

Mathlib supplies a four-stack simulation of partial-recursive program codes. A five-counter implementation, followed by prime packing and the same M2 interpreter, connects it to the scalar table. Its initial prime packing is $2^{u(x)}$, explaining the exponent in (7.1). Supplement, Section S11, describes the bridge and proof environment.

The theorem `Oloa.ThreeCounter.Program.scalar_equivalence` verifies (2.1) and (2.2) for every finite three-counter program. The closed Mathlib route uses the five-counter bridge. The paper's CRT compiler and Lean's certified search each select an integer realizing the compiled instruction word.

7.2. **Resource use.** For an already represented scalar input of L_0 bits, $T \geq 1$ continuing transitions require at most $O(TL_0+T^2)$ bit operations and $O(L_0+T)$ bits of space. Each transition increases the state length by at most 17 bits, and that one-step constant is sharp. Native input $C > 0, z \geq 0$ has exactly $17^C 13^z + 4$ bits; the three-counter route takes $z = 3^x$. These encodings therefore incur superpolynomial input-preparation costs. Supplement, Propositions S8.1 and S8.2, prove the bounds, and Section S8 discusses their computational meaning.

7.3. **Artifact release.** Release `oloa144-2026-09-28.1` supplies the mathematical supplement, one canonical data archive with the CSV, compiler, and independent numerical checker, and a separate Lean archive. `CLAIM_MAP.md` links paper results to their formal declarations or certificates; `MANIFEST.sha256` records the file fingerprints.

The Lean companion's `python3 scripts/verify.py` checks regenerated certificates against the frozen files, builds the development, audits theorem dependencies, and replays its proof environment through an official Lean kernel. The permitted logical axioms are `propext`, `Classical.choice`, and `Quot.sound`. Sections S11–S12 of the supplement give the pinned versions, verification commands, and execution records.

Appendix A. Complete numerical rule table

For $r = n \bmod 144$, continue with $(a_r n + b_r)/c_r$ when $c_r > 0$, and halt successfully at n when $c_r = 0$. The two side-by-side panels are independent residue lists. Every entry is an exact integer; the table uses no rounding. Its canonical CSV representation is included in the data and verification files; the release manifest records its fingerprint.

r	a_r	b_r	c_r	r	a_r	b_r	c_r
0	131072	9	9	72	16	-630	3
1	131072	-131054	9	73	2048	-75821	3
2	131072	-262108	9	74	16	-506	3
3	131072	-393216	9	75	256	-10029	3
4	131072	-524261	9	76	16	-685	3
5	8	5	9	77	4096	-167834	3
6	8	153	3	78	3	-42	8
7	4096	45053	3	79	9	-575	8
8	64	637	3	80	3	-12	2
9	32768	294909	3	81	3	89	2
10	8192	-81890	3	82	256	-11674	3
11	512	3581	3	83	3	647	16
12	9	324	16	84	128	-8502	3
13	9	-65	4	85	2	34	3
14	9	482	16	86	8192	-114586	3
15	9	-15	8	87	2	222	9
16	9	-80	4	88	2	-17	3
17	1	1247	16	89	2	35	3
18	81	-1422	2	90	2	927	9
19	32	-437	9	91	9	-387	16
20	2	140	9	92	9	-210	2
21	9	-105	4	93	2	-81	3
22	729	-15994	2	94	2	208	9
23	4	115	9	95	3	39	2
24	8	93	3	96	3	34	2
25	4096	45053	3	97	3	-99	8
26	64	637	3	98	3	-102	8
27	32768	294909	3	99	3	-105	8
28	2	196	9	100	3	-108	8
29	512	3581	3	101	3	625	16
30	3	116	2	102	128	-8502	3
31	9	-155	4	103	2	-110	3
32	3	-12	2	104	1	0	1
33	131072	393213	3	105	1	0	1
34	8192	-278222	9	106	1	311	3
35	3	65	2	107	2	749	9
36	1	492	8	108	16	-630	3
37	9	307	16	109	2048	-75821	3
38	1	698	16	110	16	-506	3

r	a_r	b_r	c_r	r	a_r	b_r	c_r
39	1	257	8	111	256	-10029	3
40	1	584	16	112	16	-685	3
41	1	-13	2	113	4096	-167834	3
42	2	60	3	114	1	-98	16
43	0	0	0	115	3	-153	8
44	9	-348	8	116	3	-64	2
45	16	-507	3	117	16	-324	9
46	1	610	16	118	256	-11674	3
47	3	39	2	119	16	355	9
48	3	62	2	120	256	-10029	3
49	3	27	2	121	16	-388	9
50	3	24	2	122	4096	-167834	3
51	3	21	2	123	9	-387	16
52	3	18	2	124	1	0	1
53	3	27	2	125	1	0	1
54	16	-630	3	126	128	2142	9
55	2048	-75821	3	127	256	-11674	3
56	16	-506	3	128	128	1904	9
57	256	-10029	3	129	2048	30585	9
58	16	-685	3	130	3	-58	2
59	4096	-167834	3	131	2	-91	9
60	8	93	3	132	1	-68	8
61	4096	45053	3	133	1	-61	8
62	64	637	3	134	1	-46	8
63	32768	294909	3	135	1	-3	4
64	256	-11674	3	136	9	12	2
65	512	3581	3	137	1	-81	8
66	9	-18	16	138	128	-8502	3
67	3	-153	8	139	0	0	0
68	3	-28	2	140	9	-210	2
69	131072	393213	3	141	9	-69	8
70	2	34	3	142	177147	354290	2
71	3	507	16	143	131072	131063	9

Acknowledgments

The preparation of the manuscript, computational checks, and Lean development benefited from assistance by OpenAI's AI systems. Responsibility for the mathematical claims and their presentation rests with the author.

References

[1] M. Carneiro, *Formalizing computability theory via partial recursive functions*, in 10th International Conference on Interactive Theorem Proving, Leibniz Int. Proc. Inform. **141**, Schloss Dagstuhl, 2019, 12:1–12:17. doi:10.4230/LIPIcs.ITP.2019.12.

[2] J. H. Conway, *Unpredictable iterations*, in Proceedings of the 1972 Number Theory Conference, University of Colorado, Boulder, 1972, 49–52. Reprinted in J. C. Lagarias (ed.), The Ultimate Challenge: The $3x+1$ Problem, American Mathematical Society, Providence, RI, 2010, 219–223 (with editorial commentary).

[3] J. H. Conway, *FRACTRAN: A simple universal programming language for arithmetic*, in T. M. Cover and B. Gopinath (eds.), Open Problems in Communication and Computation, Springer, New York, 1987, 4–26. doi:10.1007/978-1-4612-4808-8_2.

[4] 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.

[5] A. Dudenhefner, *Certified decision procedures for two-counter machines*, in 7th International Conference on Formal Structures for Computation and Deduction, Leibniz Int. Proc. Inform. **228**, Schloss Dagstuhl, 2022, 16:1–16:18. doi:10.4230/LIPIcs.FSCD.2022.16.

[6] Ľ. Gregušová and I. Korec, *Small universal Minsky machines*, in Mathematical Foundations of Computer Science 1979, Lecture Notes in Comput. Sci. **74**, Springer, 1979, 308–316. doi:10.1007/3-540-09526-8_28.

[7] F. Kaščák, *Small universal one-state linear operator algorithm*, in Mathematical Foundations of Computer Science 1992, Lecture Notes in Comput. Sci. **629**, Springer, 1992, 327–335. doi:10.1007/3-540-55808-X_31.

[8] I. Korec, *Small universal register machines*, Theoret. Comput. Sci. **168** (1996), no. 2, 267–301. doi:10.1016/S0304-3975(96)00080-1.

[9] D. Larchey-Wendling, *Synthetic undecidability of MSELL via FRACTRAN mechanised in Coq*, in 6th International Conference on Formal Structures for Computation and Deduction, Leibniz Int. Proc. Inform. **195**, Schloss Dagstuhl, 2021, 18:1–18:20. doi:10.4230/LIPIcs.FSCD.2021.18.

[10] The mathlib Community, *The Lean mathematical library*, in Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, ACM, 2020, 367–381. doi:10.1145/3372885.3373824.

[11] M. L. Minsky, *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, NJ, 1967.

Independent Researcher