Random Self-modifiable Computation

Michael Stephen Fiske

1 Introduction

What is computation? This question usually assumes that the Turing machine[1] [23] is the standard model [18, 19, 21]. We reexamine this question with a new model, called the *ex-machine* [12]. This model adds two special instructions to the Turing machine instructions. The name ex-machine comes from the Latin term *extra machinam* because the ex-machine computation is a non-autonomous dynamical system [10] that may no longer be considered a machine.[2] The *meta instruction* adds new states and new instructions or can replace instructions. The *random instruction* can be physically realized with a quantum random number generator [14, 15]. When an ex-machine uses meta and random instructions, its program complexity (machine size [3]) can increase, unlike a lever, pulley, or Turing machine. Two identical ex-machines can evolve to different ex-machines even when both start executing with the same tape input and initial state.

The original version of this chapter was revised: The DOI in reference 12 has been corrected. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-70873-3_72

[1]The conception of the Turing machine was motivated by Hilbert's goal to find a general method for constructing proofs of mathematical theorems [15].

[2]Each Turing machine is a discrete autonomous dynamical system in $\mathbb{C}$. See the Appendix.

M. S. Fiske (✉)
Aemea Institute, San Francisco, CA, USA
e-mail: mf@aemea.org

H. R. Arabnia et al. (eds.), *Advances in Software Engineering, Education, and e-Learning*, Transactions on Computational Science and Computational Intelligence, https://doi.org/10.1007/978-3-030-70873-3_27

We combine self-modification and randomness and construct an ex-machine $Z(x)$ whose program complexity $|Q||A|$ increases as it executes.[3] $Z(x)$'s non-autonomous behavior circumvents the contradiction in an information-theoretic proof [4, 5] of Turing's halting problem. The proof's contradiction depends upon an information-theoretic property: each Turing machine is representable with a finite number of bits that stays constant during the entire execution. Some ex-machines violate this property. $Z(x)$'s circumvention occurs because its meta instructions increase the number of states and instructions in $Z(x)$, based on random information obtained from its random instructions. Hence, the minimal number of bits that represent an ex-machine's evolved program can increase without bound as execution proceeds.

1.1 Related Work—Computation

In [24], the notion of providing an oracle was introduced. Turing stated that *an oracle cannot be a machine* but did not provide a physical basis for its existence. For a summary of various physical realizations that use quantum events to generate random binary outcomes, see [14]. In [7], the following question was asked: *Is there anything that can be done by a machine with a random element but not by a deterministic machine?* They showed for a Turing computable probability p (e.g., $p = \frac{1}{2}$) that any set of output symbols that can be enumerated with positive probability by their probabilistic machine can also be enumerated by a Turing machine. Overall, they were unable to produce Turing incomputable computation when p is Turing computable. In [13], a framework is developed for self-modifying programs, but it does not include randomness and does not address computability. In [11], a parallel machine self-modifies with meta commands and takes quantum random measurements to execute a Turing incomputable black box. Prior hypercomputation models [8, 16, 17] are not physically realizable.

2 The Ex-machine

$\mathbb{Z}$, $\mathbb{N}$, and $\mathbb{N}^+$ are the integers, non-negative integers, and positive integers, respectively. The finite set $Q = \{0, 1, 2, \ldots, n-1\} \subset \mathbb{N}$ represents the ex-machine states. As a subset of $\mathbb{N}$, Q helps specify how new states are added to Q when a meta instruction executes. Let $V = \{a_1, \ldots, a_n\}$. The set $A = \{\texttt{0}, \texttt{1}, \texttt{\#}\} \cup V$ consists of alphabet (tape) symbols, where # is the blank symbol and $\{0, 1, \#\} \cap V = \emptyset$. In some ex-machines, $A = \{\texttt{0}, \texttt{1}, \texttt{\#}, \texttt{Y}, \texttt{N}, \texttt{a}\}$, where $V = \{\texttt{Y}, \texttt{N}, \texttt{a}\}$. Sometimes, $A = \{\texttt{0}, \texttt{1}, \texttt{\#}\}$. Alphabet symbols are scanned from and written on the tape. The tape is a function $T : \mathbb{Z} \rightarrow A$. We say T is *finite* [21], whenever a finite number of tape squares $T(k)$ contain non-blank symbols.

[3] Q and A represent the ex-machine states and alphabet, respectively.

2.1 Standard Instructions

Definition 1 (Execution of Standard Instructions) Standard instructions S satisfy $S \subset Q \times A \times Q \times A \times \{-1, 0, 1\}$ and a uniqueness condition: If $(q_1, \alpha_1, r_1, a_1, y_1) \in S$ and $(q_2, \alpha_2, r_2, a_2, y_2) \in S$ and $(q_1, \alpha_1, r_1, a_1, y_1) \neq (q_2, \alpha_2, r_2, a_2, y_2)$, then $(q_1, \alpha_1) \neq (q_2, \alpha_2)$. Instruction $I = (q, a, r, \alpha, y)$ follows [19]. When the ex-machine is in state q and the tape head is scanning $a = T(k)$ at tape square k, I executes as follows. The ex-machine state moves from state q to state r. Alphabet symbol a is replaced with α so that $T(k) = \alpha$. If $y = -1$ or 1, the tape head moves one square to the left or right, respectively. If $y = 0$, the tape head does not move.

A Turing machine [23] has a finite set of machine states, a finite alphabet, a finite tape, and a finite set of standard instructions that execute according to Definition 1. An ex-machine that uses only standard instructions is called a *standard machine* and is computationally equivalent to a Turing machine. The Turing machine is the standard mathematical model of computation, realized by digital computers [1].

2.2 Random Instructions

This subsection defines two random axioms and the random instructions. Repeated independent trials are called *quantum random Bernoulli trials* [9] if all trials have a quantum random measurement [14] with only two outcomes and the probability of each outcome stays constant. *Unbiased* means that the probability of both outcomes is the same.

Random Axiom 1 (Unbiased Trials) Quantum random outcome x_i measures 0 or 1. Probability $P(x_i = 1) = P(x_i = 0) = \frac{1}{2}$.

Random Axiom 2 (Stochastic Independence) Prior measurements $x_1, \ldots, x_{i-1}$ have no effect on the next measurement x_i. For each $b_i \in \{0, 1\}$, the conditional probabilities satisfy $P(x_i = 1 | x_1 = b_1, \ldots, x_{i-1} = b_{i-1}) = \frac{1}{2}$ and $P(x_i = 0 | x_1 = b_1, \ldots, x_{i-1} = b_{i-1}) = \frac{1}{2}$.

Definition 2 (Execution of Random Instructions) Random instructions $\mathfrak{R}$ are a subset of $Q \times A \times Q \times \{-1, 0, 1\}$. $\mathfrak{R}$ satisfies uniqueness condition: If $(q_1, \alpha_1, r_1, y_1) \in \mathfrak{R}$ and $(q_2, \alpha_2, r_2, y_2) \in \mathfrak{R}$ and $(q_1, \alpha_1, r_1, y_1) \neq (q_2, \alpha_2, r_2, y_2)$, then $(q_1, \alpha_1) \neq (q_2, \alpha_2)$. When scanning symbol a and in state q, instruction (q, a, r, y) executes as follows:

(1) Measure bit $b \in \{0, 1\}$ from a quantum random source that satisfies both axioms.
(2) On the tape, an alphabet symbol a is replaced with a random bit b. Note $\{0, 1\} \subset A$.
(3) The ex-machine state q changes to state r.

(4) The tape head moves left if $y = -1$, moves right if $y = 1$, and does not move if $y = 0$.

Example 1 lists a random walk ex-machine; it shows how the random instructions execute and how the ex-machine can exhibit non-autonomous dynamical behavior.

Example 1 (Random Walk Ex-machine) Alphabet A = {0, 1, #, E}. Q = {0,1,2,3,4,5,6,h} with halting state h = 7. There are 3 random instructions (0,#,0,0), (1,#,1,0), and (4,#,4,0).

```
(0,#,0,0)       (0,0,1,0,-1)   (0,1,4,1,1)    ; Comments follow a semicolon.
(1,#,1,0)       (1,0,1,0,-1)   (1,1,2,#,1)    ; Resume random walk to the left
   of tape square 0
(2,0,3,#,1)     (2,#,h,E,0)    (2,1,h,E,0)
(3,#,0,#,-1)    ; Go back to state 0.  Number of random 0's  =  Number of
  random 1's.
(3,0,1,0,-1)    ; Go back to state 1.  Number of random 0's  >  Number of
  random 1's.
(3,1,h,E,0)
(4,#,4,0)       (4,1,4,1,1)    (4,0,5,#,-1)  ; Resume random walk to the right
  of tape square 0
(5,1,6,#,-1)    (5,#,h,E,0)    (5,0,h,E,0)
(6,#,0,#,1)     ; Go back to state 0.  Number of random 0's  =  Number of
   random 1's.
(6,1,4,1,1)     ; Go back to state 4.  Number of random 1's  >  Number of
   random 0's.
(6,0,h,E,0)
```

A valid initial tape contains only blank symbols. A valid initial state is 0. At step 1, random instruction (0,#,0,0) measures 0, so it executes (0,#,0,0,0). At step 3, instruction (1,#,1,0) measures 1, so it executes (1,#,1,1,0). (Per Definition 2, 0_r means 0 was randomly measured, and 1_r means 1 was measured.) In all executions shown, the tape head is reading the symbol to the right of the space. The sequence of tape symbols shows the tape contents after the instruction in the same row has executed.
First Execution of Random Walk Ex-machine. Steps 1–7.

STATE	TAPE	HEAD	INSTRUCTION
0	### 0###	0	(0,#,0,0_r,0)
1	## #0###	-1	(0,0,1,0,-1)
1	## 10###	-1	(1,#,1,1_r,0)
2	### 0###	0	(1,1,2,#,1)
3	#### ###	1	(2,0,3,#,1)
0	### ####	0	(3,#,0,#,-1)
0	### 0###	0	(0,#,0,0_r,0)

For the second execution, at step 1, a random measurement returns a 1, so (0,#,0,0) executes as (0,#,0,1,0). Instruction (4,#,4,0) measures 0, so (4,#,4,0,0) executes.
Second Execution of Random Walk Ex-machine. Steps 1–7.

STATE	TAPE	HEAD	INSTRUCTION
0	### 1###	0	(0,#,0,1_r,0)
4	###1 ###	1	(0,1,4,1,1)
4	###1 0##	1	(4,#,4,0_r,0)
5	### 1###	0	(4,0,5,#,-1)
6	## #####	-1	(5,1,6,#,-1)
0	### ####	0	(6,#,0,#,1)
0	### 1###	0	(0,#,0,1_r,0)

The first and second executions show that the execution behavior of the same ex-machine with identical initial conditions may be distinct at two different instances. Hence, the ex-machine is a discrete, non-autonomous dynamical system [10].

2.3 Meta Instructions

This subsection defines the meta instruction and the notion of evolving an ex-machine. The execution of a meta instruction can add new states and new instructions or replace instructions. Formally, the meta instructions $\mathfrak{M}$ satisfy $\mathfrak{M} \subset \{(q, a, r, \alpha, y, J) : q \in \mathfrak{Q}$ `and` $r \in \mathfrak{Q} \cup \{|\mathfrak{Q}|\}$ `and` $a, \alpha \in A$ `and instruction` $J \in \mathfrak{S} \cup \mathfrak{R}\}$. Define $\mathfrak{I} = \mathfrak{S} \cup \mathfrak{R} \cup \mathfrak{M}$, as the set of standard, random, and meta instructions. To help describe how a meta instruction modifies $\mathfrak{I}$, the *unique state, scanning symbol condition* is defined. For any two distinct instructions in $\mathfrak{I}$, at least one of the first two coordinates must differ. More precisely, all six of the following uniqueness conditions must hold.

1. If $(q_1, \alpha_1, r_1, \beta_1, y_1)$ `and` $(q_2, \alpha_2, r_2, \beta_2, y_2)$ both are in $\mathfrak{S}$, then $(q_1, \alpha_1) \neq (q_2, \alpha_2)$.
2. If $(q_1, \alpha_1, r_1, \beta_1, y_1) \in \mathfrak{S}$ `and` $(q_2, \alpha_2, r_2, y_2) \in \mathfrak{R}$, then $(q_1, \alpha_1) \neq (q_2, \alpha_2)$.
3. If $(q_1, \alpha_1, r_1, y_1)$ `and` $(q_2, \alpha_2, r_2, y_2)$ both are in $\mathfrak{R}$, then $(q_1, \alpha_1) \neq (q_2, \alpha_2)$.
4. If $(q_1, \alpha_1, r_1, y_1) \in \mathfrak{R}$ `and` $(q_2, \alpha_2, r_2, a_2, y_2, J_2) \in \mathfrak{M}$, then $(q_1, \alpha_1) \neq (q_2, \alpha_2)$.
5. If $(q_1, \alpha_1, r_1, \beta_1, y_1) \in \mathfrak{S}$ `and` $(q_2, \alpha_2, r_2, a_2, y_2, J_2) \in \mathfrak{M}$, then $(q_1, \alpha_1) \neq (q_2, \alpha_2)$.
6. If $(q_1, \alpha_1, r_1, a_1, y_1, J_1) \in \mathfrak{M}$ `and` $(q_2, \alpha_2, r_2, a_2, y_2, J_2) \in \mathfrak{M}$, then $(q_1, \alpha_1) \neq (q_2, \alpha_2)$.

Given a valid machine specification, conditions 1–6 assure that there is no ambiguity on what instruction to execute. The execution of a meta instruction preserves conditions 1–6.

Definition 3 (Execution of Meta Instructions) Meta instruction (q, a, r, α, y, J) executes as follows:

(1) The first five coordinates (q, a, r, α, y) are executed as a standard instruction according to Definition 1 with one caveat. State q may be expressed as $|\mathfrak{Q}|$-`c` and state r may be expressed as $|\mathfrak{Q}|$ or $|\mathfrak{Q}|$-`d`, where $0 <$ `c, d` $\leq |\mathfrak{Q}|$. When (q, a, r, α, y) is executed, if q is expressed as $|\mathfrak{Q}|$-`c` , the value of q is instantiated to the current value of $|\mathfrak{Q}| - c$. Similarly, if r is expressed as $|\mathfrak{Q}|$ or $|\mathfrak{Q}|$-`d`, the value of state r is instantiated to the current value of $|\mathfrak{Q}|$ or $|\mathfrak{Q}| - d$, respectively.

(2) Instruction J modifies $\mathfrak{I}$, where J has the form $J = (q, a, r, \alpha, y)$ or $J = (q, a, r, y)$. If $\mathfrak{I} \cup \{J\}$ satisfies the unique state, scanning symbol condition, then $\mathfrak{I}$ is updated to $\mathfrak{I} \cup \{J\}$. Otherwise, there is an instruction I in $\mathfrak{I}$ whose first

two coordinates q and a equal instruction J's first two coordinates. In this case, instruction J replaces instruction I in $\mathfrak{I}$, and $\mathfrak{I}$ is updated to $\mathfrak{I} \cup \{J\} - \{I\}$.

Remark 1 (Ex-machine Instructions are Sequences of Sets) This remark clarifies the definitions of machine states, standard, random, and meta instructions. The machine states are formally a sequence of sets. When the notation is formally precise, the machine states are expressed as $\mathfrak{Q}(m)$, where m indicates that the mth computational step has executed. The standard, random, and all ex-machine instructions are also sequences of sets, represented as $\mathfrak{S}(m)$, $\mathfrak{R}(m)$, and $\mathfrak{I}(m)$, respectively. Usually, index m is not shown in expressions $\mathfrak{Q}$, $\mathfrak{S}$, $\mathfrak{R}$, $\mathfrak{M}$, or $\mathfrak{I}$.

Example 2 shows how to add an instruction to $\mathfrak{I}$ and how to instantiate new states in $\mathfrak{Q}$.

Example 2 (Adding New States and Instructions) Consider a meta instruction $(q, a_1, |\mathfrak{Q}|\texttt{-1}, \alpha_1, y_1, J)$, where $J = (|\mathfrak{Q}|\texttt{-1}, a_2, |\mathfrak{Q}|, \alpha_2, y_2)$. After instruction $(q, a_1, |\mathfrak{Q}|\texttt{-1}, \alpha_1, y_1)$ executes, this meta instruction adds a new state $|\mathfrak{Q}|$ to the states $\mathfrak{Q}$ and adds instruction J, instantiated with the current value of $|\mathfrak{Q}|$. For clarity, states are red and alphabet symbols are blue. Set $\mathfrak{Q} = \{ 0, 1, 2, 3, 4, 5, 6, 7\}$. Set $A = \{ \texttt{\#}, \texttt{0}, \texttt{1}\}$. An initial configuration is shown below.

```
State       Tape
  5        ##11 01##
```

Meta instruction $(5, \texttt{0}, |\mathfrak{Q}| - 1, \texttt{1}, 0, J)$ executes with values $q = 5$, $a_1 = \texttt{0}$, $\alpha_1 = \texttt{1}$, $y_1 = 0$, $a_2 = \texttt{1}$, $\alpha_2 = \texttt{\#}$, and $y_2 = -1$. Note $J = (|\mathfrak{Q}|\texttt{-1}, \texttt{1}, |\mathfrak{Q}|, \texttt{\#}, -1)$. Since $|\mathfrak{Q}| = 8$, instruction $(5, \texttt{0}, 7, \texttt{1}, 0)$ is executed. Also, standard instruction $J = (7, \texttt{1}, 8, \texttt{\#}, -1)$ is added as a new instruction. The instantiation of $|\mathfrak{Q}| = 8$ in J adds state 8; the states are updated to $\mathfrak{Q} = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$. After $(5, \texttt{0}, |\mathfrak{Q}| - 1, \texttt{1}, 0, J)$ executes, the new ex-machine configuration is shown below.

```
State       Tape
  7        ##11 11##
```

Now, the ex-machine is scanning a $\texttt{1}$ and lying in state 7, so the standard instruction $J = (7, \texttt{1}, 8, \texttt{\#}, -1)$ executes. (Note that J was just added to the instructions.) After J executes, the new configuration is shown below.

```
State       Tape
  8        ##1 1#1##
```

Remark 2 (Self-reflection of $|\mathfrak{Q}|$ *)* Consider an ex-machine $\mathfrak{X}$ with a meta instruction I containing symbol $|\mathfrak{Q}|$. The instantiation of $|\mathfrak{Q}|$ invokes *self-reflection* about $\mathfrak{X}$'s current number of states, at the moment when $\mathfrak{X}$ executes I. This type of self-reflection can be physically realized.

Definition 4 (Simple Meta Instructions) $\texttt{(q,a,}|\mathfrak{Q}|\texttt{-d,b,y)}$, $\texttt{(q,a,}|\mathfrak{Q}|\texttt{,b,y)}$, $\texttt{(}|\mathfrak{Q}|\texttt{-c,a,r,y)}$, $\texttt{(}|\mathfrak{Q}|\texttt{-c,a,}|\mathfrak{Q}|\texttt{-d,b,y)}$, or $\texttt{(}|\mathfrak{Q}|\texttt{-c,a,}|\mathfrak{Q}|\texttt{,b,y)}$ are valid expressions for simple meta instructions, where $0 < \texttt{c}, \texttt{d} \leq |\mathfrak{Q}|$. Symbols $|\mathfrak{Q}|\texttt{-c}$, $|\mathfrak{Q}|\texttt{-d}$, and $|\mathfrak{Q}|$ instantiate to a state based on the value of $|\mathfrak{Q}|$ when the simple meta instruction executes.

Herein, ex-machines self-reflect only with symbols $|\mathfrak{Q}|\texttt{-1}$ and $|\mathfrak{Q}|$.

Example 3 (Execution of Simple Meta Instructions.) $A = \{$`0`,`1`,`#`$\}$ and $\mathfrak{Q} = \{0\}$. Instructions ($|\mathfrak{Q}|$`-1,#,`$|\mathfrak{Q}|$`-1,1,0`) ($|\mathfrak{Q}|$`-1,1,`$|\mathfrak{Q}|$`,0,1`)

```
STATE    TAPE     HEAD     INSTRUCTION       NEW INSTRUCTION
  0      # 1##     0       (0,#,0,1,0)       (0,#,0,1,0)
  1      #0 ##     1       (0,1,1,0,1)       (0,1,1,0,1)
  1      #0 1#     1       (1,#,1,1,0)       (1,#,1,1,0)

  2      #00 #     2       (1,1,2,0,1)       (1,1,2,0,1)
```

With an initial blank tape and starting state of 0, four computational steps are shown above. At step 1, $\mathfrak{X}$ scans `#` and lies in state 0. Since $|\mathfrak{Q}| = 1$, a simple meta instruction ($|\mathfrak{Q}|$`-1,#,`$|\mathfrak{Q}|$`-1,1,0`) instantiates to `(0,#,0,1,0)` and executes. At step 2, $\mathfrak{X}$ scans `1` and lies in state 0. Since $|\mathfrak{Q}| = 1$, ($|\mathfrak{Q}|$`-1,1,`$|\mathfrak{Q}|$`,0,1`) instantiates to `(0,1,1,0,1)`, updates $\mathfrak{Q} = \{0, 1\}$, and executes `(0,1,1,0,1)`.

Definition 5 (Finite Initial Conditions) Ex-machine $\mathfrak{X}$ has finite initial conditions if the 4 conditions hold before $\mathfrak{X}$'s instructions are executed: (1) The number of states $|\mathfrak{Q}|$ is finite. (2) The number of alphabet symbols $|A|$ is finite. (3) The number of instructions $|\mathfrak{J}|$ is finite. (4) The tape is finite.

An ex-machine's initial conditions are analogous to a differential equation's boundary value conditions. Remark 3 assures that the ex-machine computation is physically plausible.

Remark 3 (Finite Initial Conditions) If the machine starts its execution with finite initial conditions, then after the machine has executed l instructions for any positive integer l, the current number of states $\mathfrak{Q}(l)$ is finite and the current set of instructions $\mathfrak{J}(l)$ is finite. Also, tape T is still finite, and the number of quantum random measurements obtained is finite.

Proof The execution of one meta instruction adds at most one new instruction and one new state to $\mathfrak{Q}$. Using induction, Remark 3 follows from Definitions 1, 2, 3, and 5.

An ex-machine can evolve from a prior computation. *Evolution* is useful: random and meta instructions can increase an ex-machine's complexity via self-modification.

Definition 6 (Evolving an Ex-machine) Let T_0, T_1, $T_2 \dots T_{i-1}$ each be a finite tape. Consider an ex-machine $\mathfrak{X}_0$ with finite initial conditions. $\mathfrak{X}_0$ starts executing with tape T_0 and evolves to ex-machine $\mathfrak{X}_1$ with tape S_1 after the execution halts. Subsequently, $\mathfrak{X}_1$ starts executing with tape T_1 and evolves to $\mathfrak{X}_2$ with tape S_2. This means that when ex-machine $\mathfrak{X}_1$ starts executing on tape T_1, its instructions are preserved after the halt with tape S_1. The ex-machine evolution continues until $\mathfrak{X}_{i-1}$ starts executing with tape T_{i-1} and evolves to ex-machine $\mathfrak{X}_i$ with tape S_i after the execution halts. One says that the ex-machine $\mathfrak{X}_0$ evolves to $\mathfrak{X}_i$ after i halts.

When $\mathfrak{X}_0$ evolves to $\mathfrak{X}_1$, then $\mathfrak{X}_1$ evolves to $\mathfrak{X}_2$, and so on up to $\mathfrak{X}_n$, then $\mathfrak{X}_i$ is an *ancestor* of $\mathfrak{X}_j$ if $0 \le i < j \le n$. Similarly, $\mathfrak{X}_j$ is a *descendant* of $\mathfrak{X}_i$ when $i < j$. The sequence of ex-machines $\mathfrak{X}_0 \to \mathfrak{X}_1 \to \cdots \to \mathfrak{X}_n \dots$ is an *evolutionary path.*

3 Computing Ex-machine Languages

A class of ex-machines are evolutions of a fundamental ex-machine $\mathfrak{Z}(x)$, whose 15 instructions are listed in Definition 9. These ex-machines compute languages L that are subsets of $\{\mathtt{a}\}^* = \{\mathtt{a}^n : n \in \mathbb{N}\}$. $\mathtt{a}^n$ stands for n a's. The empty string is $\mathtt{a}^0$ and $\mathtt{a}^3 = \mathtt{aaa}$. Set language space $\mathfrak{L} = \bigcup\limits_{L \subset \{a\}^*} \{L\}$. Function $f : \mathbb{N} \to \{0, 1\}$ defines language L_f.

Definition 7 (Language L_f) $f : \mathbb{N} \to \{0, 1\}$ induces language $L_f = \{\mathtt{a}^n : f(n) = 1\}$. String $\mathtt{a}^n$ is in L_f iff $f(n) = 1$.

Trivially, L_f is a language in $\mathfrak{L}$. Moreover, these functions f generate all of $\mathfrak{L}$.

Remark 4 (Language Space) $\mathfrak{L} = \bigcup\limits_{f \in \{0,1\}^{\mathbb{N}}} \{L_f\}$.

Definition 8 ($\mathfrak{X}$ Computes Language L in $\mathfrak{L}$) Set alphabet $A = \{\#, \mathtt{0}, \mathtt{1}, \mathtt{N}, \mathtt{Y}, \mathtt{a}\}$. Let $\mathfrak{X}$ be an ex-machine. The language L in $\mathfrak{L}$ that $\mathfrak{X}$ computes is defined as follows. A valid initial tape has the form $\#\ \ \#\mathtt{a}^n\#$. The valid initial tape $\#\ \ \#\#$ represents the empty string. After $\mathfrak{X}$ starts executing with initial tape $\#\ \ \#\mathtt{a}^n\#$, string $\mathtt{a}^n$ is in $\mathfrak{X}$'s language if $\mathfrak{X}$ halts with tape $\#\mathtt{a}^n\#\ \ \mathtt{Y}\#$. String $\mathtt{a}^n$ is not in $\mathfrak{X}$'s language if $\mathfrak{X}$ halts with tape $\#\mathtt{a}^n\#\ \ \mathtt{N}\#$.

The use of special alphabet symbols Y and N—to decide whether $\mathtt{a}^n$ is in the language—follows [18]. For string $\#\ \ \#\mathtt{a}^m\#\ $, some $\mathfrak{X}$ could first halt with $\#\mathtt{a}^m\#\ \ \mathtt{N}\#$ and in a second execution could halt with $\#\mathtt{a}^m\#\ \ \mathtt{Y}\#$. The oscillation of halting outputs can continue indefinitely, and $\mathfrak{X}$'s language is not well defined per Definition 8. In this chapter, we avoid ex-machines whose halting outputs do not stabilize.

3.1 *Ex-machine $\mathfrak{Z}(x)$*

The purpose of Definition 9 is to show that $\mathfrak{Z}(x)$ can evolve to compute any language L_f in $\mathfrak{L}$; and that evolutions of $\mathfrak{Z}(x)$ compute Turing incomputable languages on a set of Lebesgue measure 1 in language space $\mathfrak{L}$, where $\mathfrak{L}$ also has measure 1.

Definition 9 (Ex-machine $\mathfrak{Z}(x)$) $A = \{\#, \mathtt{0}, \mathtt{1}, \mathtt{N}, \mathtt{Y}, \mathtt{a}\}$. States $\mathfrak{Q} = \{\ \mathtt{0}, \mathtt{h}, \mathtt{n}, \mathtt{y}, \mathtt{t}, \mathtt{v}, \mathtt{w}, \mathtt{x}, \mathtt{8}\ \}$ where halting state $\mathtt{h} = 1$ and states $\mathtt{n} = 2$, $\mathtt{y} = 3$, $\mathtt{t} = 4$, $\mathtt{v} = 5$, $\mathtt{w} = 6$, $\mathtt{x} = 7$. The initial state is always 0. For the reader's benefit, letters represent states instead of explicit numbers. State n indicates NO that the string is not in the language. State y indicates YES that the string is in the language. State x helps generate a new random bit.

```
(0,#,8,#,1)       (8,#,x,#,0)
(y,#,h,Y,0)       (n,#,h,N,0)
(x,#,x,0)         (x,a,t,0)
```

($|\mathfrak{Q}|-1$,a,x,a,0)
($|\mathfrak{Q}|-1$,#,x,#,0)

(x,0,v,#,0,($|\mathfrak{Q}|-1$,#,n,#,1))
(x,1,w,#,0,($|\mathfrak{Q}|-1$,#,y,#,1))

(t,0,w,a,0,($|\mathfrak{Q}|-1$,#,n,#,1))
(t,1,w,a,0,($|\mathfrak{Q}|-1$,#,y,#,1))

(v,#,n,#,1,($|\mathfrak{Q}|-1$,a,$|\mathfrak{Q}|$,a,1))
(w,#,y,#,1,($|\mathfrak{Q}|-1$,a,$|\mathfrak{Q}|$,a,1))
(w,a,$|\mathfrak{Q}|$,a,1,($|\mathfrak{Q}|-1$,a,$|\mathfrak{Q}|$,a,1))

With initial state 0 and tape # #aaaa##, an execution instance of $\mathfrak{Z}(x)$ is below.

STATE	TAPE	HEAD	INSTRUCTION EXECUTED	NEW INSTRUCTION
8	# aaaa###	1	(0,#,8,#,1)	
x	# aaaa###	1	(8,a,x,a,0)	(8,a,x,a,0)
t	# 1aaa###	1	(x,a,t,1_r,0)	
w	# aaaa###	1	(t,1,w,a,0,($|\mathfrak{Q}|-1$,#,y,#,1))	(8,#,y,#,1)
9	#a aaa###	2	(w,a,$|\mathfrak{Q}|$,a,1,($|\mathfrak{Q}|-1$,a,$|\mathfrak{Q}|$,a,1))	(8,a,9,a,1)
x	#a aaa###	2	(9,a,x,a,0)	(9,a,x,a,0)
t	#a 1aa###	2	(x,a,t,1_r,0)	
w	#a aaa###	2	(t,1,w,a,0,($|\mathfrak{Q}|-1$,#,y,#,1))	(9,#,y,#,1)
10	#aa aa###	3	(w,a,$|\mathfrak{Q}|$,a,1,($|\mathfrak{Q}|-1$,a,$|\mathfrak{Q}|$,a,1))	(9,a,10,a,1)
x	#aa aa###	3	(10,a,x,a,0)	(10,a,x,a,0)
t	#aa 0a###	3	(x,a,t,0_r,0)	
w	#aa aa###	3	(t,0,w,a,0,($|\mathfrak{Q}|-1$,#,n,#,1))	(10,#,n,#,1)
11	#aaa a###	4	(w,a,$|\mathfrak{Q}|$,a,1,($|\mathfrak{Q}|-1$,a,$|\mathfrak{Q}|$,a,1))	(10,a,11,a,1)
x	#aaa a###	4	(11,a,x,a,0)	(11,a,x,a,0)
t	#aaa 1###	4	(x,a,t,1_r,0)	
w	#aaa a###	4	(t,1,w,a,0,($|\mathfrak{Q}|-1$,#,y,#,1))	(11,#,y,#,1)
12	#aaaa ###	5	(w,a,$|\mathfrak{Q}|$,a,1,($|\mathfrak{Q}|-1$,a,$|\mathfrak{Q}|$,a,1))	(11,a,12,a,1)
x	#aaaa ###	5	(12,#,x,#,0)	(12,#,x,#,0)
x	#aaaa 0##	5	(x,#,x,0_r,0)	
v	#aaaa ###	5	(x,0,v,#,0,($|\mathfrak{Q}|-1$,#,n,#,1))	(12,#,n,#,1)
n	#aaaa# ##	6	(v,#,n,#,1,($|\mathfrak{Q}|-1$,a,$|\mathfrak{Q}|$,a,1))	(12,a,13,a,1)
h	#aaaa# N#	6	(n,#,h,N,0)	

This instance of $\mathfrak{Z}(x)$'s execution replaces (8,#,x,#,0) with (8,#,y,#,1). Instruction (w,a,$|\mathfrak{Q}|$,a,1,($|\mathfrak{Q}|$-1,a,|Q|,a,1)) replaces (8,a,x,a,0) with new instruction (8,a,9,a,1). Also, the simple meta instruction (|Q|-1,a,x,a,0) temporarily added instructions (9,a,x,a,0), (10,a,x,a,0), and (11,a,x,a,0). These instructions are replaced by (9,a,10,a,1), (10,a, 11,a,1), and (11,a 12,a,1), respectively. Instruction (|Q|-1,#,x,#,0) added (12,#,x,#,0) and instruction (12,#,n,#,1) replaced (12,#,x, #,0). Instructions (9,#,y,#,1), (10,#,n,#,1), (11,#,y,#,1), and (12,a,13,a,1) are added. Five new states 9, 10, 11, 12, and 13 are added to $\mathfrak{Q}$. After halting, $\mathfrak{Q} = \{$0, h, n, y, t, v, w, x, 8, 9, 10, 11, 12, 13$\}$, and the evolved ex-machine $\mathfrak{Z}(11010\,x)$ has 24 instructions.

Two different instances of $\mathfrak{Z}(x)$ can evolve to two different machines and compute distinct languages according to Definition 8. After $\mathfrak{Z}(x)$ has evolved to a new machine $\mathfrak{Z}(a_0a_1 \dots a_m\, x)$ as a result of a prior execution with input tape # #a^m#, then for each i with $0 \le i \le m$, machine $\mathfrak{Z}(a_0a_1 \dots a_m\, x)$ always halts with the same output when presented with input tape # #a^i#. $\mathfrak{Z}(a_0a_1 \dots a_m\, x)$'s halting output stabilizes on all input strings a^i where $0 \le i \le m$. Example 4 shows this stabilization property.

Example 4 (Ex-machine $\mathfrak{Z}(1101\,x)$)

(0,#,8,#,1) (y,#,h,Y,0) (n,#,h,N,0)

(x,#,x,0) (x,a,t,0)
(x,0,v,#,0,($|\mathfrak{Q}|-1$,#,n,#,1))
(x,1,w,#,0,($|\mathfrak{Q}|-1$,#,y,#,1))

(t,0,w,a,0,($|\mathfrak{Q}|-1$,#,n,#,1))
(t,1,w,a,0,($|\mathfrak{Q}|-1$,#,y,#,1))

(v,#,n,#,1,($|\mathfrak{Q}|-1$,a,$|\mathfrak{Q}|$,a,1))
(w,#,y,#,1,($|\mathfrak{Q}|-1$,a,$|\mathfrak{Q}|$,a,1))
(w,a,$|\mathfrak{Q}|$,a,1,($|\mathfrak{Q}|-1$,a,$|\mathfrak{Q}|$,a,1))

($|\mathfrak{Q}|-1$,a,x,a,0)
($|\mathfrak{Q}|-1$,#,x,#,0)

```
(8,#,y,#,1)     (8,a,9,a,1)
(9,#,y,#,1)     (9,a,10,a,1)
(10,#,n,#,1)    (10,a,11,a,1)
(11,#,y,#,1)    (11,a,12,a,1)
(12,#,n,#,1)    (12,a,13,a,1)
```

New instructions (8,#,y,#,1), (9,#,y,#,1), and (11,#,y,#,1) help $\mathfrak{Z}(11010\,x)$ compute that the empty strings a and aaa are in its language, respectively. Similarly, the new instructions (10,#,n,#,1) and (12,#,n,#,1) help $\mathfrak{Z}(11010\,x)$ compute that aa and aaaa are not in its language, respectively. The 1's in $\mathfrak{Z}(11010\,x)$'s name indicate that the empty strings a and aaa are in its language. The 0's indicate that strings aa and aaaa are not in its language. Symbol x indicates that $\mathfrak{Z}(11010\,x)$ has not yet determined for $n \geq 5$ whether strings $\mathtt{a}^n$ are in $\mathfrak{Z}(11010\,x)$'s language.

Starting at state 0, $\mathfrak{Z}(11010\,x)$ computes that the empty string is in its language

```
STATE     TAPE        HEAD      INSTRUCTION
  8       ## ###        1       (0,#,8,#,1)
  y       ### ##        2       (8,#,y,#,1)
  h       ### Y#        2       (y,#,h,Y,0)
```

Starting at state 0, $\mathfrak{Z}(11010\,x)$ computes that string aa is not in its language.

```
STATE     TAPE        HEAD      INSTRUCTION
  8       ## aa###      1       (0,#,8,#,1)
  9       ##a a###      2       (8,a,9,a,1)
 10       ##aa ###      3       (9,a,10,a,1)
  n       ##aa# ##      4       (10,#,n,#,1)
  h       ##aa# N#      4       (n,#,h,N,0)
```

Similarly, starting at state 0, $\mathfrak{Z}(11010\,x)$ computes that a and aaa are in its language and $\mathfrak{Z}(11010\,x)$ computes that aaaa is not in its language. For each of these executions, no new states are added and no instructions are added or replaced. Thus, for all subsequent executions, $\mathfrak{Z}(11010\,x)$ computes that the empty strings a and aaa are in its language, and strings aa and aaaa are not.

Starting at state 0, below is an execution of $\mathfrak{Z}(11010\,x)$ on input tape # #aaaaaa##.

STATE	TAPE	HEAD	INSTRUCTION EXECUTED	NEW INSTRUCTION
8	# aaaaaa##	1	(0,#,8,#,1)	
9	#a aaaaa##	2	(8,a,9,a,1)	
10	#aa aaaa##	3	(9,a,10,a,1)	
11	#aaa aaa##	4	(10,a,11,a,1)	
12	#aaaa aa##	5	(11,a,12,a,1)	
13	#aaaaa a##	6	(12,a,13,a,1)	
x	#aaaaa a##	6	(13,a,x,a,0)	
t	#aaaaa 0##	6	(x,a,t,0_r,0)	
w	#aaaaa a##	6	(t,0,w,a,0,($\lvert\mathfrak{Q}\rvert-1$,#,n,#,1))	(13,#,n,#,1)
14	#aaaaaa ##	7	(w,a,$\lvert\mathfrak{Q}\rvert$,a,1,($\lvert\mathfrak{Q}\rvert-1$,a,$\lvert\mathfrak{Q}\rvert$,a,1))	(13,a,14,a,1)
x	#aaaaaa ##	7	(14,#,x,#,0)	(14,#,x,#,0)
x	#aaaaaa 1#	7	(x,#,x,1_r,0)	
w	#aaaaaa ##	7	(x,1,w,#,0,($\lvert\mathfrak{Q}\rvert-1$,#,y,#,1))	(14,#,y,#,1)
y	#aaaaaa# #	8	(w,#,y,#,1,($\lvert\mathfrak{Q}\rvert-1$,a,$\lvert\mathfrak{Q}\rvert$,a,1))	(14,a,15,a,1)
h	#aaaaaa# Y	8	(y,#,h,Y,0)	

$\mathfrak{Z}(11010\,x)$ evolves to $\mathfrak{Z}(11010\,01\,x)$. The first random instruction (x,a,t,0) measures a 0, so it executes as (x,a,t, 0_r,0). Instruction (13,#,n,#,1) is added due to the random 0 bit; in all subsequent executions of $\mathfrak{Z}(11010\,01\,x)$, string a^5 is not in $\mathfrak{Z}(11010\,01\,x)$'s language. The second random instruction (x,#,x,0) measures a 1 and executes as (x,#,x,1_r,0). Instruction (14,#,y,#,1) is added. In all subsequent executions, string a^6 is in $\mathfrak{Z}(11010\,01\,x)$'s language.

Definition 10 specifies $\mathfrak{Z}(a_0a_1\dots a_m\,x)$ and covers $\mathfrak{Z}(11010\,x)$'s execution.

Definition 10 (Ex-machine $\mathfrak{Z}(a_0a_1\dots a_m\,x)$) Let $m\in\mathbb{N}$. Set $\mathfrak{Q}=\{$0, h, n, y, t, v, w, x, 8, 9, 10, … $m+8, m+9\}$. For $0\le i\le m$, a_i is 0 or 1. In $\mathfrak{Z}(a_0a_1\dots a_m\,x)$'s instructions, symbol $b_8=$ y if $a_0=1$, else $b_8=$ n if $a_0=0$; symbol $b_9=$ y if $a_1=1$, else $b_9=$ n if $a_1=0$; and so on until the second to the last instruction ($m+8$,#,b_{m+8},#,1), $b_{m+8}=$ y if $a_m=1$, else $b_{m+8}=$ n if $a_m=0$.

(0,#,8,#,1) (y,#,h,Y,0) (n,#,h,N,0)

(x,#,x,0)
(x,a,t,0)

($\lvert\mathfrak{Q}\rvert-1$,a,x,a,0)
($\lvert\mathfrak{Q}\rvert-1$,#,x,#,0)

(x,0,v,#,0,($\lvert\mathfrak{Q}\rvert-1$,#,n,#,1))
(x,1,w,#,0,($\lvert\mathfrak{Q}\rvert-1$,#,y,#,1))

(t,0,w,a,0,($\lvert\mathfrak{Q}\rvert-1$,#,n,#,1))
(t,1,w,a,0,($\lvert\mathfrak{Q}\rvert-1$,#,y,#,1))

(v,#,n,#,1,($\lvert\mathfrak{Q}\rvert-1$,a,$\lvert\mathfrak{Q}\rvert$,a,1))
(w,#,y,#,1,($\lvert\mathfrak{Q}\rvert-1$,a,$\lvert\mathfrak{Q}\rvert$,a,1))
(w,a,$\lvert\mathfrak{Q}\rvert$,a,1,($\lvert\mathfrak{Q}\rvert-1$,a,$\lvert\mathfrak{Q}\rvert$,a,1))

(8,#, b_8,#,1) (8,a,9,a,1) (9,#,b_9,#,1) (9,a,10,a,1)
(10,#, b_{10},#,1) (10,a,11,a,1) . . . ($i+8$,#,b_{i+8},#,1) ($i+8$,a,$i+9$,a,1) . . .
($m+7$,#,b_{m+7},#,1) ($m+7$,a,$m+8$,a,1) ($m+8$,#,b_{m+8},#,1) ($m+8$,a,$m+9$,a,1)

Lemma 1 *If i satisfies $0 \le i \le m$, string $\mathtt{a}^i$ is in $\mathfrak{Z}(a_0 a_1 \ldots a_m\, x)$'s language if $a_i = 1$, and string $\mathtt{a}^i$ is not in $\mathfrak{Z}(a_0 a_1 \ldots a_m\, x)$'s language if $a_i = 0$. If $n > m$, it has not yet been determined whether $\mathtt{a}^n$ is in $\mathfrak{Z}(a_0 a_1 \ldots a_m\, x)$'s language or not in its language.*

Proof When $0 \le i \le m$, the first consequence follows immediately from the definition of $\mathtt{a}^i$ being in $\mathfrak{Z}(a_0 a_1 \ldots a_m\, x)$'s language and from Definition 10. In instruction `(`$i+8$`,#,`b_{i+8}`,#,1)`, the state value of b_{i+8} is `y` if $a_i = 1$ and b_{i+8} is `n` if $a_i = 0$.

For the indeterminacy of strings $\mathtt{a}^n$ when $n > m$, $\mathfrak{Z}(a_0 \ldots a_m\, x)$ executes its last instruction `(`$m+8$`,a,` $m+9$`,a,1)` when scanning the mth `a` in $\mathtt{a}^n$. For each `a` to the right of `#`$\mathtt{a}^m$ on the tape, $\mathfrak{Z}(a_0 \ldots a_m\, x)$ executes random instruction `(x,a,t,0)`.

If `(x,a,t,0)` measures `0`, then meta instructions `(t,0,w,a,0,(|Ω|-1,#,n,#,1))` and `(w,a,|Ω|,a,1 (|Ω|-1,a,|Ω|,a,1))` execute. Otherwise, `(x,a,t,0)` measures `1`, so `(t,1,w,a,0, (|Ω|-1,#,y,#,1))` and `(w,a,|Ω|,a,1, (|Ω|-1,a,|Ω|,a, 1))` execute. If the next alphabet symbol to the right is an `a`, then a new standard instruction executes, derived from an instantiation of `(|Ω|-1,a,x,a,0)`. When the tape head scans the last `a` in $\mathtt{a}^n$, a new standard instruction executes, derived from `(|Ω|-1,#,x,#,0)`.

For each `a` to the right of `#`$\mathtt{a}^m$ on the tape, the execution of random instruction `(x,a,t,0)` determines whether string $\mathtt{a}^{m+k}$, such that $1 \le k \le n - m$, is in $\mathfrak{Z}(a_0 a_1 \ldots a_n\, x)$'s language. After the execution of `(|Ω|-1,#,x,#,0)`, the tape head is scanning a blank symbol, so the random instruction `(x,#,x,0)` is executed. If the random source generates `0`, the meta instructions `(x,0,v,#,0, (|Ω|-1,#,n,#,1))` and `(v,#,n,#,1, (|Ω|-1,a,|Ω|, a, 1))` execute. Then, instruction `(n,#,h,N,0)` executes last, which indicates that $\mathtt{a}^n$ is not in $\mathfrak{Z}(a_0 a_1 \ldots a_n\, x)$'s language. If the execution of `(x,#,x,0)` measures `1`, the instructions `(x,1,w,#,0,(|Ω|-1,#,y,#,1))` and `(w,#,y,#, 1,(|Ω| -1,a,|Ω|,a,1))` execute. Then, instruction `(y,#,h,Y,0)` executes last, which indicates that $\mathtt{a}^n$ is in $\mathfrak{Z}(a_0 a_1 \ldots a_n\, x)$'s language. During the execution of the instructions, for each `a` on the tape to the right of `#`$\mathtt{a}^m$, $\mathfrak{Z}(a_0 a_1 \ldots a_m\, x)$ evolves to $\mathfrak{Z}(a_0 a_1 \ldots a_n\, x)$ according to the instructions, specified by Definition 10, where one substitutes n for m.

3.2 *Some Turing Incomputable Properties of $\mathfrak{Z}(x)$*

When the measurements in $\mathfrak{Z}(x)$'s two random instructions satisfy both axioms, all 2^n finite paths of length n in the infinite binary tree of Fig. 1 are equally likely. The 1-to-1 correspondence between $f : \mathbb{N} \to \{0, 1\}$ and an infinite downward path (red) in the binary tree helps show that $\mathfrak{Z}(x)$ can evolve to compute any language L_f in $\mathfrak{L}$.

Consider $\mathfrak{Z}(x)$ and all $\mathfrak{Z}(a_0 \ldots a_m\, x)$ for each $m \in \mathbb{N}$ and $a_0 \ldots a_m$ in $\{0, 1\}^{m+1}$.

Theorem 1 *Each language L_f in $\mathfrak{L}$ can be computed by the evolving sequence of ex-machines $\mathfrak{Z}(x)$, $\mathfrak{Z}(f(0)\, x)$, $\mathfrak{Z}(f(0) f(1)\, x)$, ..., $\mathfrak{Z}(f(0) f(1) \ldots f(n)\, x)$, ...*

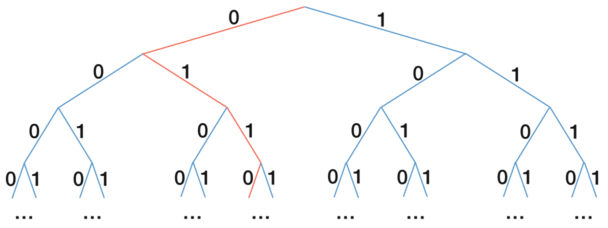

Fig. 1 Infinite binary tree. A graphical representation of $\{0, 1\}^{\mathbb{N}}$

Proof Apply Definitions 9 and 10 and Lemma 1.

Corollary 1 *For any* $f : \mathbb{N} \rightarrow \{0, 1\}$ *and any* n*, the evolving sequence* $\mathfrak{Z}(f(0)\,x)$*,* $\ldots \mathfrak{Z}(f(0)f(1)\ldots f(n)f(n+1)\,x)$*, ... computes language* L_f*.*

Corollary 2 *For each* n*, the evolution of ex-machines* $\mathfrak{Z}(x)$*,* $\mathfrak{Z}(f(0)x)$*,* $\mathfrak{Z}(f(0)f(1)\,x)$*, ...,* $\mathfrak{Z}(f(0)f(1)\ldots f(n)\,x)$ *have cumulatively used only a finite amount of tape, finite number of states, finite number of instructions, and finite number of instruction executions, and only a finite amount of quantum information is measured by the random instructions.*

Proof Remark 3 and Definitions 5 and 10 imply finite computational resources.

Theorem 2 and Corollary 3 come from the following intuition. A set X is countable if there exists a bijection between X and $\mathbb{N}$. $\mathfrak{L}$ is uncountable, so most languages L_f in $\mathfrak{L}$ are Turing incomputable. Since each L_f is equally likely of being computed by $\mathfrak{Z}(x)$, most languages computed by $\mathfrak{Z}(x)$'s evolution are Turing incomputable.

For each $n \in \mathbb{N}$, define language tree $\mathfrak{L}(a_0 \ldots a_n) = \{L_f : f \in \{0, 1\}^{\mathbb{N}}$ and $f(i) = a_i$ for i such that $0 \leq i \leq n\}$. Define subtree $\mathfrak{S}(a_0 \ldots a_n) = \{f \in \{0, 1\}^{\mathbb{N}} : f(i) = a_i$ such that $0 \leq i \leq n\}$. Let Ψ be this 1-to-1 correspondence: $\mathfrak{L} \overset{\Psi}{\leftrightarrow} \{0, 1\}^{\mathbb{N}}$ and $\mathfrak{L}(a_0 \ldots a_n) \overset{\Psi}{\leftrightarrow} \mathfrak{S}(a_0 \ldots a_n)$. Since random axioms 1 and 2 hold, each finite path $f(0)f(1)\ldots f(n)$ is equally likely. There are 2^{n+1} of these paths, so each path has probability $2^{-(n+1)}$. The uniform probabilities on finite strings of the same length extend to Lebesgue [9, 22] measure μ on probability space $\{0, 1\}^{\mathbb{N}}$. Subtree $\mathfrak{S}(a_0 \ldots a_n)$ has measure $2^{-(n+1)}$, where $\mu\big(\mathfrak{S}(a_0 \ldots a_n)\big) = 2^{-(n+1)}$ and $\mu(\{0, 1\}^{\mathbb{N}}) = 1$. Via Ψ, μ induces uniform probability measure ν on $\mathfrak{L}$, where $\nu\big(\mathfrak{L}(a_0 \ldots a_n)\big) = 2^{-(n+1)}$ and $\nu(\mathfrak{L}) = 1$.

Theorem 2 *The Turing incomputable languages* L_f *have measure 1 in* $(\nu, \mathfrak{L})$*.*

Proof The Turing computable functions $f : \mathbb{N} \rightarrow \{0, 1\}$ are countable. Via the Ψ correspondence, the Turing computable languages L_f have ν-measure 0 in $\mathfrak{L}$.

Corollary 3 *For all* $a_0 \ldots a_m$ *in* $\{0, 1\}^{m+1}$*,* $\mathfrak{Z}(a_0 \ldots a_m\,x)$ *is not a Turing machine.*

Proof $\mathfrak{Z}(x)$ can evolve to compute Turing incomputable languages on a set of ν-measure 1 in $\mathfrak{L}$. $\mathfrak{Z}(a_0\;\ldots a_m\,x)$ can evolve to compute Turing incomputable languages on a set of ν-measure $2^{-(m+1)}$ in $\mathfrak{L}$. Each Turing machine only computes one language, so the measure of all Turing computable languages is 0 in $\mathfrak{L}$.

4 An Ex-machine Halting Problem

In [23], Turing posed the question, does there exist a Turing machine $\mathfrak{D}$ that can determine for any given Turing machine M and finite tape T whether M's execution on tape T eventually halts? Turing proved that no Turing machine could solve this problem. His halting problem can be extended to ex-machines. Does there exist an ex-machine $\mathfrak{X}(x)$ such that for any given Turing machine M, then $\mathfrak{X}(x)$ can sometimes compute whether M's execution on finite initial tape T will eventually halt? In order for this question to be well-posed, the phrase *can sometimes compute whether* must be defined.

From the universal Turing machine / enumeration theorem [21], there is a Turing computable enumeration $\mathfrak{E} : \mathbb{N} \rightarrow \{\texttt{Turing machines } M\} \times \{\texttt{Each initial state of } M\}$ of every Turing machine. Similar to ex-machines, for each machine M, the set $\{\texttt{Each initial state of } M\}$ is realized as a finite subset $\{0, \ldots, n-1\}$ of $\mathbb{N}$. Since $\mathfrak{E}(n)$ is an ordered pair, the phrase "Turing machine $\mathfrak{E}(n)$" refers to the first coordinate of $\mathfrak{E}(n)$. The "initial state $\mathfrak{E}(n)$" refers to the second coordinate of $\mathfrak{E}(n)$. Turing's halting problem is equivalent to the blank-tape halting problem [19]. The blank-tape halting problem translates to: for each Turing machine $\mathfrak{E}(n)$, does $\mathfrak{E}(n)$ halt when $\mathfrak{E}(n)$ begins executing with a blank initial tape and initial state $\mathfrak{E}(n)$?

Lemma 1 implies that the same initial ex-machine can evolve to two different ex-machines; these two ex-machines will never compute the same language no matter what descendants they evolve to. For example, $\mathfrak{Z}(0\,x)$ and $\mathfrak{Z}(1\,x)$ can never compute the same language in $\mathfrak{L}$. Hence, *sometimes* means that for each n, there exists an evolution of $\mathfrak{X}(x)$ to $\mathfrak{X}(a_0 x)$, then to $\mathfrak{X}(a_0 a_1 x)$, and so on up to $\mathfrak{X}(a_0 a_1 \ldots a_n\ x)$ $\ldots$, where for each i with $0 \le i \le n$, then $\mathfrak{X}(a_0 a_1 \ldots a_n\ x)$ correctly computes whether Turing machine $\mathfrak{E}(n)$ halts or does not halt. The word *computes* means that $\mathfrak{X}(a_0 a_1 \ldots a_i\ x)$ halts after a finite number of instructions executed, and the halting output written by $\mathfrak{X}(a_0 a_1 \ldots a_i\ x)$ on the tape indicates whether machine $\mathfrak{E}(n)$ halts. For example, if the input tape is `#  #a`^{i}`#`, then enumeration machine $M_{\mathfrak{E}}$ writes the representation of $\mathfrak{E}(i)$ on the tape, and then $\mathfrak{X}(a_0 a_1 \ldots a_m\, x)$ with $m \ge i$ halts with `#  Y#` written to the right of the representation for machine $\mathfrak{E}(i)$. Alternatively, $\mathfrak{X}(a_0 a_1 \ldots a_m\, x)$ with $m \ge i$ halts with `#  N#` written to the right of the representation for machine $\mathfrak{E}(i)$. The word *correctly* means that ex-machine $\mathfrak{X}(a_0 a_1 \ldots a_m\, x)$ halts with `#  Y#` written on the tape if machine $\mathfrak{E}(i)$ halts and ex-machine $\mathfrak{X}(a_0 a_1 \ldots a_m\, x)$ halts with `#  N#` written on the tape if machine $\mathfrak{E}(i)$ does not halt.

Next, the ex-machine halting problem is transformed so that the results from Sect. 3 can be applied. Choose alphabet $\mathfrak{A} = \{$`#, 0, 1, a, A, B, M, N, S, X, Y`$\}$.

As before, identify the set of Turing machine states $\mathfrak{Q}$ as a finite subset of $\mathbb{N}$. Let $M_{\mathfrak{E}}$ be the Turing machine that computes a Turing computable enumeration [4] as $\mathfrak{E}_a : \mathbb{N} \to \{\mathfrak{A}\}^* \times \mathbb{N}$, where the tape # #a^n# represents natural number n. Each $\mathfrak{E}_a(n)$ is an ordered pair where the first coordinate is the Turing machine and the second coordinate is an initial state chosen from $\mathfrak{E}_a(n)$'s states. Define the *halting function* $h_{\mathfrak{E}_a} : \mathbb{N} \to \{0, 1\}$ such that for each n, set $h_{\mathfrak{E}_a}(n) = 1$, whenever $\mathfrak{E}_a(n)$ halts. Otherwise, set $h_{\mathfrak{E}_a}(n) = 0$, if $\mathfrak{E}_a(n)$ with blank initial tape and initial state $\mathfrak{E}_a(n)$ does not halt. Function $h_{\mathfrak{E}_a}(n)$ is well defined because for each $n \in \mathbb{N}$, with blank initial tape and initial state $\mathfrak{E}_a(n)$, Turing machine $\mathfrak{E}_a(n)$ either halts or does not halt. Via function $h_{\mathfrak{E}_a}(n)$ and Definition 7, define *halting language* $L_{h_{\mathfrak{E}_a}}$.

Theorem 3 *There exists an evolutionary path for ex-machine $\mathfrak{Z}(x)$ that computes halting language $L_{h_{\mathfrak{E}_a}}$; namely, $\mathfrak{Z}(h_{\mathfrak{E}_a}(0)\ x) \to \mathfrak{Z}(h_{\mathfrak{E}_a}(0)\ h_{\mathfrak{E}_a}(1)\ x) \to \ldots$ $\mathfrak{Z}(h_{\mathfrak{E}_a}(0)\ h_{\mathfrak{E}_a}(1) \ldots h_{\mathfrak{E}_a}(m)\ x) \ldots$*

Proof Apply the mathematical developments in the previous three paragraphs, using halting function $h_{\mathfrak{E}_a}$, language $L_{h_{\mathfrak{E}_a}}$, and Theorem 1.

Theorem 3 implies that a proof by contradiction for Turing's halting problem does not hold for ex-machines: the existence of path $\mathfrak{Z}(h_{\mathfrak{E}_a}(0)\ x) \to \mathfrak{Z}(h_{\mathfrak{E}_a}(0)\ h_{\mathfrak{E}_a}(1)\ x) \to \ldots\ \mathfrak{Z}(h_{\mathfrak{E}_a}(0)\ h_{\mathfrak{E}_a}(1) \ldots h_{\mathfrak{E}_a}(m)\ x) \ldots$ circumvents the contradiction. From an information-theoretic perspective, almost every (w.r.t. to μ on $\{0, 1\}^{\mathbb{N}}$) evolutionary path $\mathfrak{Z}(f(0)\ x) \to \mathfrak{Z}(f(0)f(1)\ x) \to \ldots\ \mathfrak{Z}(f(0)f(1) \ldots f(n)\ x) \ldots$ avoids the contradiction in Chaitin's information-theoretic proof [5] that the halting problem for Turing machines is unsolvable. The contradiction depends upon the following: the minimum number of bits needed to represent a Turing machine stays constant. In contrast, there is a set $\mathfrak{F} \subset \{0, 1\}^{\mathbb{N}}$ with $\mu(\mathfrak{F}) = 1$ such that for all $f \in \mathfrak{F}$, the minimum number of bits needed to represent $\mathfrak{Z}(f(0)f(1) \ldots f(n)\ x)$ increases without bound as n increases.

5 A Research Direction

Theorem 3 and information-theoretic analysis both show that a proof by contradiction of the unsolvability of Turing's halting problem does not apply to ex-machines. This capability suggests that novel self-modification procedures, cleverly integrated with randomness, should be explored to help enhance theorem proving [2, 6] and constructive type systems that use conservative workarounds [20] to avoid the halting problem.

[4]Chapter 7 of [19] provides explicit details of encoding quintuples with a particular universal Turing machine. Alphabet $\mathfrak{A}$ was selected to be compatible with this encoding. A careful study of chapter 7 provides a clear path of how $M_{\mathfrak{E}}$'s instructions can be specified to implement $\mathfrak{E}_a$.

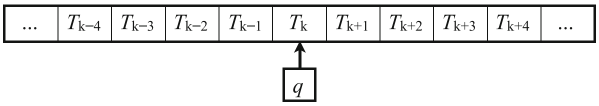

Fig. 2 Machine configuration (q, k, T) before executing a standard instruction

6 Conclusion

We showed that ex-machines can compute Turing incomputable languages and that ex-machines are not limited by the halting problem for Turing machines. The language computed by an ex-machine reflects its computational capabilities. The problem of determining program correctness for a digital computer program is unsolvable by a Turing machine. The detection of an infinite loop in a computer program (i.e., a case of program correctness) can be reduced to Turing's halting problem. For these reasons, it is important to understand how far methods of evaluating program correctness for digital computer programs can be extended with randomness and advanced self-modification procedures.

Appendix: A Turing Machine Is an Autonomous Dynamical System

Fix a Turing machine M. Transformation ϕ maps a machine configuration to a point in the complex plane $\mathbb{C}$; ϕ also maps each of M's instructions to a finite number $|A|$ of unique affine functions each with a distinct domain. These affine functions can be extended to a function F on a bounded region W in $\mathbb{C}$, containing these domains and a disjoint bounded set, called the halting attractor. Via ϕ, one computational step of M corresponds to one iteration of the discrete autonomous dynamical system (F, W).

Let machine states $\mathsf{Q} = \{q_1, \ldots, q_m\}$. Let alphabet $A = \{a_1, \ldots, a_n\}$, where a_1 is the blank symbol. Halt state h is a special state that is not in Q. Function $\eta : \mathsf{Q} \times A \to \mathsf{Q} \cup \{h\} \times A \times \{-1, +1\}$ is the machine M's program. A single instruction is $\eta(q, a) = (r, b, x)$, where $q \in \mathsf{Q}$, $r \in \mathsf{Q} \cup \{h\}$, $a, b \in A$, and $x \in \{-1, +1\}$. Set $B = |A| + |\mathsf{Q}| + 1$. Define symbol value function $\nu : \{h\} \cup \mathsf{Q} \cup A \to \mathbb{N}$ as $\nu(a_1) = 0, \ldots, \nu(a_i) = i - 1, \ldots, \nu(a_n) = |A| - 1$, $\nu(h) = |A|$, $\nu(q_1) = |A| + 1$, $\ldots, \nu(q_i) = |A| + i, \ldots, \nu(q_m) = |A| + |\mathsf{Q}|$.

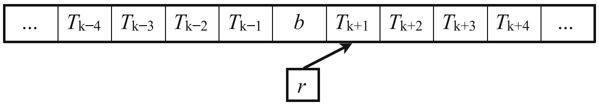

Fig. 3 Machine configuration after executing instruction $\eta(q, T_k) = (r, b, +1)$

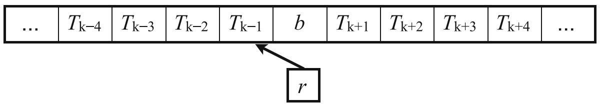

Fig. 4 Machine configuration after executing instruction $\eta(q, T_k) = (r, b, -1)$

$T : \mathbb{Z} \to A$ is the tape and is finite. T_k is the alphabet symbol in tape square k. Machine configuration (q, k, T) lies in $\mathsf{Q} \times \mathbb{Z} \times A^{\mathbb{Z}}$ and maps to the complex number:

$$\phi(q, k, T) = |A|\nu(T_k) + \sum_{j=0}^{\infty} \nu(T_{k+j+1})|A|^{-j} + \Big(B\nu(q) + \sum_{j=0}^{\infty} \nu(T_{k-j-1})|A|^{-j}\Big)\, i. \tag{1}$$

In Eq. 1, the infinite series in both the real and imaginary parts sums to rational numbers because the initial tape squares contain a finite number of non-blank symbols.

Next, we define how ϕ maps each instruction in program η to a finite set of affine functions. When instruction $\eta(q, T_k) = (r, b, +1)$ executes, state q moves to state r, symbol b replaces T_k on tape square k, and the head moves to tape square $k + 1$.

The right affine functions corresponding to instruction $\eta(q, T_k) = (r, b, +1)$ are of the form $f(x + yi) = f_1(x) + f_2(y)\, i$, where $f_1(x) = |A|x + m$ and $f_2(y) = \frac{1}{|A|} y + n$. Using Eq. 1 and Fig. 3 to solve for m and n, ϕ maps instruction $\eta(q, T_k) = (r, \alpha, +1)$ to the affine function $f(x + yi) = f_1(x) + f_2(y)\, i$, where

$$f_1(x) = |A|x + (|A| - 1)\nu(T_{k+1}) - |A|^2\nu(T_k) \tag{2}$$

$$f_2(y) = \frac{1}{|A|} y + B\nu(r) + \nu(b) - \frac{B}{|A|}\nu(q). \tag{3}$$

For each of the $|A|$ distinct values $v(T_{k+1})$ in f_1, f is a different affine function. Thus, there are $|A|$ distinct affine functions that correspond to instruction $\eta(q, T_k) = (r, b, +1)$. The domain of each right affine function is $U_{j,k} = \big\{x + yi \in \mathbb{C} : j \le x < j + 1 \text{ and } k \le y < k + |A|\big\}$, where $j = |A|\nu(T_k) + \nu(T_{k+1})$ and $k = B\nu(q)$.

When $\eta(q, T_k) = (r, b, -1)$ executes, state q moves to state r, symbol b replaces T_k on square k, and the head moves to tape square $k - 1$.

From Eq. 1 and Fig. 4, ϕ maps instruction $\eta(q, T_k) = (r, b, -1)$ to affine function $g(x + yi) = g_1(x) + g_2(y)\, i$, where

$$g_1(x) = \frac{1}{|A|} x + |A|\nu(T_{k-1}) + \nu(b) - \nu(T_k) \tag{4}$$

$$g_2(y) = |A|y + B\nu(r) - |A|B\nu(q) - |A|\nu(T_{k-1}). \tag{5}$$

For each of the $|A|$ distinct values $v(T_{k-1})$ in g_1 and g_2, g is a different affine function. Thus, there are $|A|$ distinct left affine functions that correspond to instruction $\eta(q, T_k) = (r, b, -1)$. The domain of each left affine function is $V_{j,k} = \{x + yi \in \mathbb{C} : j \le x < j + |A| \texttt{ and } k \le y < k + 1\}$, where $j = |A|\nu(T_k)$ and $k = B\nu(q) + \nu(T_{k-1})$.

Define *halting attractor* $H = \{x + yi \in \mathbb{C} : 0 \le x < |A|^2 \texttt{ and } B|A| \le y \le (B + 1)|A|\}$. The points in $\mathbb{C}$ that correspond to halting configurations (h, k, T) are called *halting points*. Using elementary algebra and simple geometric series calculations, one can verify that the halting points are a subset of H. Define *halting map* $\mathfrak{h} : H \to H$, where $\mathfrak{h}(x + yi) = x + yi$ on H. Every point in the halting attractor is a fixed point of $\mathfrak{h}$. Moreover, the intersection of each affine function's domain and H is empty. This implies that $\mathfrak{h}$ and all left and right affine functions corresponding to η's instructions can be extended to a function F on domain W that contains H and all domains $U_{j,k}$ and $V_{j,k}$.

Overall, the ϕ *correspondence transforms Turing's halting problem to a discrete autonomous dynamical systems problem* in $\mathbb{C}$. If machine configuration (q, k, T) halts after n computational steps, then the orbit of $\phi(q, k, T)$ exits one of the domains $U_{j,k}$ or $V_{j,k}$ on the nth iteration and enters the halting attractor H. If machine configuration (r, j, S) never halts, then the orbit of $\phi(r, j, S)$ never reaches the halting attractor.

References

1. H. Abelson, G.J. Sussman, J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd edn. (MIT Press, Cambridge, 1996)
2. Y. Bertot, P. Castéran, *Interactive Theorem Proving & Program Development* (Springer, Berlin, 2004)
3. M. Blum, On the size of machines. Inf. Control **11**, 257–265 (1967)
4. C. Calude, *Information and Randomness* (Springer, Berlin, 2002), pp. 362–363
5. G. Chaitin. *Information, Randomness, and Incompleteness* (World Scientific, Singapore, 1987)
6. T. Coquand, G. Huet, Calculus of constructions. Inf. Comput. **76**, 95–120 (1988)
7. K. de Leeuw, E.F. Moore, C.E. Shannon, N. Shapiro, Computability by probabilistic machines, in ed. by Shannon & McCarthy *Automata Studies* (Princeton University Press, Princeton, 1956), pp. 183–212
8. G. Etesi, I. Nemeti, Non-Turing computations via Malament-Hogarth spacetimes. Int. J. Theoret. Phys. **41**(2), 341–370 (2002)
9. W. Feller, *Introduction to Probability Theory and Its Applications*, vol. 1 (Wiley, Hoboken, 1957), vol. 2 (1966)
10. M.S. Fiske, *Non-autonomous Dynamical Systems Applicable to Neural Computation* (Northwestern University, Evanston, 1996)
11. M.S. Fiske, Turing incomputable computation. *Turing-100. The Alan Turing Centenary.* EasyChair **10**, 66–91 (2012). https://doi.org/10.29007/x5g2
12. M.S. Fiske, Quantum random self-modifiable computation. Logic Colloquium 2019. Prague, Czech Republic, August 11–16. Bull. Symb. Logic. **25**(4), 510–511 (2019). https://doi.org/10.1017/bsl.2019.56
13. H. Godfroy, J.Y. Marion, *Abstract Self Modifying Machines* (HAL CCSD, Lyon, 2016)
14. M. Herrero-Collantes, J.C. Garcia-Escartin, Quantum random number generators. Rev. Modern Phys. **89**(1), 015004, (2017). https://arxiv.org/abs/1604.03304

15. D. Hilbert, Mathematische probleme. Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematische-Physikalische Klasse **3**, 253–297 (1900)
16. M. Hogarth, Non-turing computers and Non-turing computability, in *Proceedings of the Biennial Meeting of the Philosophy of Science Assoc.*, vol. 1 (University of Chicago, Chicago, 1994), pp. 126–138
17. T. Kieu, Quantum algorithm for Hilbert's tenth problem. Int. J. Theoret. Phys. **42**, pp. 1461–1478 (2003)
18. H.R. Lewis, C. Papadimitriou, *Elements of the Theory of Computation* (Prentice-Hall, Upper Saddle River, 1981)
19. M. Minsky, *Computation: Finite and Infinite Machines* (Prentice-Hall, Upper Saddle River, 1967), pp. 132–155
20. B. Pierce, *Types and Programming Languages* (MIT Press, Cambridge, 2002), pp. 99–100
21. H. Rogers. *Theory of Recursive Functions and Effective Computability* (MIT Press, Cambridge, 1987)
22. H.L. Royden, *Real Analysis* (Prentice-Hall, Upper Saddle River, 1988)
23. A.M. Turing, On computable numbers, with an application to the Entscheidungsproblem. Proc. London Math. Soc. Series 2. **42** (3, 4), 230–265 (1936)
24. A.M. Turing, System of logic based on ordinals. Proc. London Math. Soc. Series 2. **45**, 161–228 (1939)