# AT THE LIMITS OF COMPUTABILITY: STEPS TOWARDS AUTONOMOUS AI DISCOVERY ON THE BUSY BEAVER FRONTIER

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Recent breakthroughs in AI for mathematics—ranging from Olympiad medal performances to resolving long-standing open conjectures—have rapidly expanded what automated systems can achieve. Yet these successes primarily concern predefined, human-formulated problems, leaving open how AI systems can autonomously uncover mathematical structure and formulate new questions. To push beyond static challenges, we explore the Busy Beaver (BB) domain, an uncomputable frontier that is fundamentally open-ended. Because no algorithm can compute the Busy Beaver number $\mathrm{BB}(n)$ for every $n$, this domain provides a natural setting for sustained exploration through mathematical conjecture and proof. In this paper, we introduce a Gemini-based agentic pipeline that automatically discovers accelerations of Turing machines, expressed as compact mathematical rules summarizing many execution steps. Our system integrates pattern recognition, automated conjecturing, informal mathematical reasoning, and formal theorem proving in Lean. With this framework, our system autonomously discovered and formally verified novel Collatz-like accelerations for five machines in $\mathrm{BB}(6)$. Subject to community acceptance, we provisionally classify these machines as *Cryptids*, namely machines whose blank-tape executions admit exact, compact mathematical descriptions but whose halting behavior remains unresolved. These results advance the analysis of $\mathrm{BB}(6)$ holdouts on the Busy Beaver frontier and demonstrate the potential of autonomous AI systems for open-ended mathematical discovery in fundamental computer science.

## 1 INTRODUCTION

Recent milestones in AI-driven mathematics—from an AI system earning an IMO medal (Hubert et al., 2026) to formalizing Fermat's Last Theorem in Lean 4 or to OpenAI's multi-agent system proposing a solution to the Navier-Stokes existence and smoothness problem (OpenAI, 2026)—have largely focused on using AI to *solve* pre-defined, human-formulated questions. Gowers' distinction between problem-solving and theory-building (Gowers, 2000), together with the emphasis on mathematical understanding in the Leiden Declaration and Tao's *Mathematics in the Age of AI* (Tao, 2026), motivates a broader goal for mathematical AI: contributing to the discovery of mathematical structures and the formulation of new questions. Motivated by this broader goal, our work explores a complementary paradigm: using AI to autonomously uncover and formalize mathematical problems arising from an open-ended stream of computation. The ultimate testbed for this kind of open-ended discovery lies at the frontier of computability theory: the Busy Beaver Challenge.

Introduced by Rado (1962), the challenge poses a simple question: among all halting Turing Machines (TMs) of a given size, what is the maximum number of steps a machine can run before halting? For machines with $n$ states and two symbols started on a blank tape, this maximum is denoted by $\mathrm{BB}(n)$. Knowing $\mathrm{BB}(n)$ provides a universal halting criterion: if an $n$-state machine runs for $\mathrm{BB}(n) + 1$ steps, it runs forever. Consequently, $n \mapsto \mathrm{BB}(n)$ is uncomputable, as computability would make the Halting Problem decidable. Moreover, the Busy Beaver sequence—$\mathrm{BB}(1), \mathrm{BB}(2), \ldots$—grows faster than any computable sequence, outstripping stacked exponentials, the Ackermann function, and $\mathrm{TREE}(n)$ (Aaronson, 2020).



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Table 1: Collatz-like accelerations discovered by our system. For machines with known or suspected accelerations, we cite the original source and highlight the added value of our analysis, such as a formal proof or a higher-level acceleration. For previously unanalyzed machines, our results represent complete, novel discoveries. We distinguish three discovery types. Full: new acceleration and proof. Acceleration: new acceleration for a known machine. Proof: formal proof of rules similar to known ones. Discovered transition rules are given in Appendix B.

| Machine | BB domain | Discovery type | Comment |
|---|---|---|---|
| **Part I: New Discoveries** | | | |
| 1RB1LD_1RC0RA_1LA1RE_0LA0LB_0RB1RF_---1RD | $\mathrm{BB}(6)$ | Full | - |
| 1RB0LD_1LC1RA_1RA1LC_1LE1LA_0RF1LD_---1RE | $\mathrm{BB}(6)$ | Full | - |
| 1RB1RD_0LC1RA_0RA1LC_1RA1LE_1RF0LD_---0RE | $\mathrm{BB}(6)$ | Full | Rules include a recursive arithmetic function. |
| 1RB1RB_1LC0RE_---1LD_0RA1LB_1RA1LF_0LE1LE | $\mathrm{BB}(6)$ | Full | - |
| 1RB0LF_1LC1RA_0RE0RD_---1LE_1LF1RC_1LC1LA | $\mathrm{BB}(6)$ | Full | - |
| **Part II: Re-Discoveries** | | | |
| 1RB3LA1LA1RA1RA_2LB2RA---4RB1LB | $\mathrm{BB}(2,5)$ | Acceleration | First analyzed by mxdys on 12 Mar 2025, appears to be a Cryptid. (Wiki) |
| 1RB0RB_1LB0LC_1RD1LC_1RE0RA_0RF1RA_---1LB | $\mathrm{BB}(6)$ | Proof | The result is similar to a comment on Discord by the user Sheep. We provide proofs. |
| 1RB---_0RC0RD_1LD1RB_0LE0LC_1RA0LF_1LD1LE | $\mathrm{BB}(6)$ | Proof | Analysis by Shawn Ligocki (Wiki). |
| 1RB1RF_0LC0RF_1RD1LC_---0LE_0RC1LF_1RA0LE | $\mathrm{BB}(6)$ | Proof | The result is similar to a comment on Discord by Shawn Ligocki. We provide proofs. |

An intriguing consequence of the uncomputability of the Busy Beaver numbers is that mathematical creativity remains open-ended (Chaitin, 2013). Its significance extends beyond pure computation: major conjectures—such as Fermat's Last Theorem, the Riemann hypothesis, Goldbach's conjecture, and ZF consistency—can be encoded as the non-halting of specific Turing machines (Aaronson, 2020). Because no algorithm can compute $\mathrm{BB}(n)$ for all $n$, this frontier provides continuous scope for conjecture, proof, and new analytical techniques. This makes it an ideal testing ground for autonomous AI discovery: regardless of their sophistication, future AI systems will always encounter Busy Beaver problems beyond the reach of any given algorithm.

In practice, searching for Busy Beaver values involves applying *deciders* to vast spaces of Turing machines, progressively filtering them down to smaller sets of *holdouts* with unknown halting status. Here, we focus on a subset of approximately $1{,}000$ holdouts in $\mathrm{BB}(6)$ (Table 2). Resolving a holdout requires proving that the machine halts or runs forever. For certain holdouts, however, mathematical analysis reveals dynamics that mirror open mathematical problems; such machines are termed *Cryptids* (Definition 2.1), with many governed by Collatz-like systems. While $\mathrm{BB}(5)$ was recently resolved through community collaboration (Blanchard et al., 2026), subsequent frontiers—$\mathrm{BB}(6)$, $\mathrm{BB}(2,5)$, and $\mathrm{BB}(3,3)$—harbor Cryptids that currently prevent full classification. Given the notorious difficulty of the Collatz conjecture—which led Paul Erdős to remark that "Mathematics is not yet ready for such problems"—the community has largely pivoted toward isolating Cryptids and evaluating Busy Beaver values strictly across non-Cryptid candidates. Against this backdrop, our work focuses on accelerating machine execution. An acceleration summarizes sequences of low-level transitions into compact rules, drastically speeding simulation while exposing underlying arithmetic structure. In turn, these accelerations facilitate halting proofs or expose direct mappings to Collatz-like dynamics.

To execute this strategy, we introduce an automated, Gemini-based **coding scientist** that compresses simulated execution data into structured mathematical equations and formally proves their equiva-

Table 2: The Busy Beaver domain (Aug 2026). The initial search space of $(2m(n+1))^{mn}$ Turing machines is reduced to a set of holdouts via canonization and deciders. Our pipeline kicks in and provides candidates from the holdouts as well as verified accelerations.

| | | Community Analysis | | Our Pipeline | |
|---|---|---|---|---|---|
| **Domain** | **Machines** | **Holdouts** | **Cryptids** | **Candidates** | **New Cryptids** |
| $\mathrm{BB}(6)$ | $2.32 \times 10^{17}$ | 1094 | 18 | 51 | 5 |
| $\mathrm{BB}(2,5)$ | $5.9 \times 10^{14}$ | 65 | 2 | 2 | 0 |
| $\mathrm{BB}(3,3)$ | $2.64 \times 10^{12}$ | 4 | 1 | 0 | 0 |

lence in Lean. Rather than statically analyzing Turing machine transition rules, we treat machines as "grey boxes"—maintaining full visibility into internal states during execution while remaining agnostic to their underlying code. Relying purely on observed execution makes our approach general; in principle, it applies to any Turing-complete language, including Python and C++.[1]

Figure 1 outlines our four-stage pipeline. First, a custom compression algorithm performs *pattern detection* on execution traces, parsing data into symbolic blocks. Second, in *automated conjecturing*, a Large Language Model (LLM) acts as a mathematical explorer: it synthesizes transition rules from empirical patterns, querying an interactive simulator to resolve edge cases and construct Collatz-like transition systems. Conjectures surviving strict empirical filters advance to *informal proving*, where the LLM uses an iterative generator-evaluator loop to draft and refine step-by-step proofs against targeted verification checks. Finally, in *formal verification*, an autonomous agent interacts with Lean 4 to formalize and compile these proofs, mathematically certifying the machine's long-term dynamics.

Table 2 summarizes our findings alongside the status of frontier domains $\mathrm{BB}(6)$, $\mathrm{BB}(2,5)$, and $\mathrm{BB}(3,3)$. During conjecturing, our pipeline autonomously flagged 53 candidates: machines displaying uncharacterized Collatz-like dynamics empirically verified over their first 5 million steps. Our main result concerns five $\mathrm{BB}(6)$ holdouts with no prior documented analysis. For each, we prove in Lean that its blank-tape execution is exactly described by an explicit Collatz-like system, so that the machine halts if and only if this system reaches a halting configuration. These are new, formally verified reductions of each machine's halting problem to an arithmetic question. Subject to community acceptance, we provisionally classify these machines as Cryptids (New Cryptids column). For four further machines previously studied informally, we provide formal proofs of similar rules or new accelerations. Table 1 details all nine machines, and Appendix B gives their full transition systems.

## 2 Background

### 2.1 Turing Machines and the Busy Beaver Problem

A Turing machine is a theoretical computational model featuring a finite state set and an infinite one-dimensional tape partitioned into discrete cells. Each cell holds a symbol from a finite alphabet, conventionally $\{0,1\}$. In the Busy Beaver framework, the tape is initialized blank (all cells containing 0). The machine operates via a read/write head observing the current cell: guided by a deterministic transition function, it overwrites the symbol, shifts left or right, and updates its state. This process iterates until the machine reaches an explicit halt state, concluding computation.

The Busy Beaver problem, introduced by Tibor Radó in 1962 (Rado, 1962), asks for the maximum number of steps an $n$-state, $m$-symbol halting Turing machine can execute on an initially blank tape. This maximum is captured by the Busy Beaver function, $\mathrm{BB}(n,m)$, conventionally abbreviated to $\mathrm{BB}(n)$ for the 2-symbol case. Undecidability of the Halting Problem renders $\mathrm{BB}(n)$ fundamentally uncomputable. Proving $\mathrm{BB}(n) = B$ requires exhibiting an $n$-state machine halting at exactly $B$ steps while showing that all other $n$-state candidates either halt within $B$ steps or run forever. Machines with undetermined halting behavior from a blank tape are designated *holdouts*. Consequently, evaluating $\mathrm{BB}(n)$ entails resolving every holdout—either by extracting its runtime or certifying non-

[1] BB numbers can be formulated for any Turing-complete language by asking: "What is the largest natural number generated by a halting program of length $n$ bits?" (Chaitin, 2013)

termination. Collaborative efforts have determined several initial values: $\mathrm{BB}(2) = 6$, $\mathrm{BB}(3) = 21$, $\mathrm{BB}(4) = 107$, and recently $\mathrm{BB}(5) = 47,176,870$ (Blanchard et al., 2026). This exhaustive search leaves a frontier of complex, unresolved holdouts in expanded spaces such as $\mathrm{BB}(6)$, $\mathrm{BB}(3,3)$, and $\mathrm{BB}(2,5)$. For additional background, see the BB Wiki, Quanta Magazine, or surveys such as Michel (2009).

### 2.2 Cryptids and Collatz-like Systems

**Definition 2.1** (Cryptid)**.** In this work, a *Cryptid* is a holdout machine whose blank-tape execution admits an exact, compact description as a mathematical dynamical system, but for which determining whether the trajectory reaches a halting condition remains open. Unlike a general holdout, a Cryptid features an explicit mathematical formulation exposing the structure underlying its unresolved halting behavior.

This definition is broader than the community convention, which recognizes a machine as a Cryptid only if its associated mathematical formulation is considered genuinely difficult and open. Accordingly, our classification of newly identified candidates remains subject to community consensus.

Cryptid dynamics frequently mirror *Collatz-like* systems. This structure emerges when tape configurations are grouped into blocks of identical symbols whose variable lengths are tracked by integer exponents. Machine evolution between parametrized configurations can then be formulated as exact macro-transitions acting on these exponents. The resulting dynamics are typically piecewise affine: the update applied to each exponent depends on its congruence class (e.g., parity). As in the canonical $3x + 1$ map, modular conditions select different linear updates. Beyond revealing high-level arithmetic structure, this abstraction accelerates simulation by replacing sequences of base-level steps with single macro-transitions. Nevertheless, concise rules do not ensure predictability: for Cryptids, deciding whether trajectories ever terminate remains a difficult open question and the primary bottleneck to completing the corresponding Busy Beaver classification.

A concrete example of a Cryptid is *Antihydra*, a six-state, two-symbol candidate in the $\mathrm{BB}(6)$ domain. Consider the parametrized configurations

$$A(a,b) := 0^{\infty}\ 1^{a}\ 0\ 1^{b}\ \mathrm{E}{>}\ 0^{\infty},$$

where E$>$ records head position and internal state. Antihydra follows the exact macro-transitions:

$$A(a,2b) \xrightarrow{2a+3b^2+12b+11} A(a+2,3b+2),$$

$$A(0,2b+1) \xrightarrow{3b^2+9b-1} 0^{\infty}\ {<}\mathrm{F}\ 110\ 1^{3b}\ 0^{\infty},$$

$$A(a+1,2b+1) \xrightarrow{3b^2+12b+5} A(a,3b+3),$$

with transition arrow labels denoting base-level execution steps. Throughout, $a, b \geq 0$ range over non-negative integers, so the left-hand side acts as a pattern: for instance, $A(a+1,2b+1)$ matches any configuration whose first exponent is positive and whose second exponent is odd. Initialized on a blank tape, Antihydra reaches $A(0,4)$ after 11 steps and proceeds via these rules. The state F configuration marks termination, as the head subsequently triggers the undefined F0 transition. Thus, Antihydra terminates if and only if its trajectory from $A(0,4)$ encounters a configuration of the form $A(0,2b+1)$—a question that remains unresolved.

## 3 Method

Our automated discovery pipeline, illustrated in Figure 1, operates across four sequential stages, progressing from empirical traces to formal mathematical verification. First, *Pattern Detection* simulates machine execution to isolate recurring abstract tape configurations. Second, *Automated Conjecturing* tasks an LLM with conjecturing parameterized transition rules, iteratively refined against targeted simulation checks. Third, *Informal Proving* constructs structured mathematical proofs certifying these conjectured rules. Finally, *Formal Theorem Proving* translates informal arguments into machine-checked Lean 4 theorems, mathematically verifying the discovered long-term dynamics. We detail each stage below.

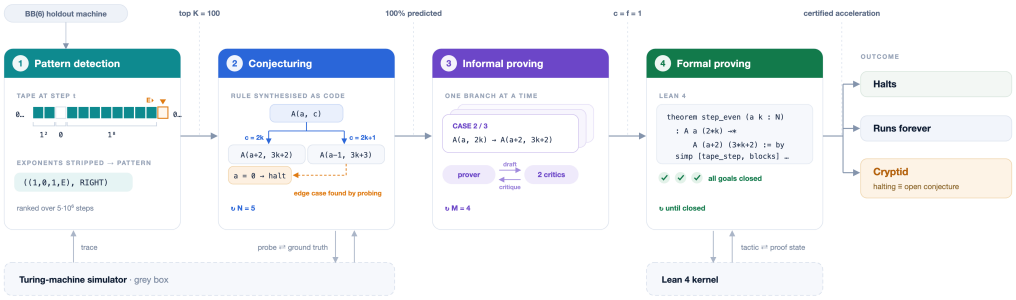



Figure 1: Overview of the automated discovery pipeline, detailing the progression from empirical pattern detection and LLM-guided rule conjecturing to informal proving and formal verification in Lean 4.

### 3.1 Pattern Detection

The first stage of our pipeline aims to empirically identify repeating patterns in the execution trace of a given Turing machine. We simulate the machine for a fixed $S$ steps, recording the exact configuration of the machine, which includes the raw contents of the tape, the active state, and the position of the read/write head.

To convert these configurations into tractable mathematical objects, we apply a custom compression algorithm. For example, a tape trace expressed in directed head notation as $0^{\infty}\ 11011111111\ \mathrm{E}{>}\ 0^{\infty}$ (where E is the active state and $>$ denotes orientation) is parsed into exponential symbol blocks: $0^{\infty}\ 1^2\,0\,1^8\ \mathrm{E}{>}\ 0^{\infty}$. The algorithm parses finite tape segments via greedy search, evaluating a simplicity metric—the ratio of repetitions to block size—to favor compact, highly repetitive sequences over sparser ones. It incorporates a one-step lookahead heuristic to handle boundary ambiguities, discarding a greedy repetition when doing so yields greater overall downstream compression. Full details and pseudocode are given in Appendix C.

Once a configuration is compressed, we abstract it into a *family of configurations*, which we refer to simply as a *pattern*. A pattern is defined by the sequence of block symbols, the current machine state, and the direction of the read/write head; crucially, the specific block exponents are stripped away. Following the previous example, $0^{\infty}\ 1^2\,0\,1^8\ \mathrm{E}{>}\ 0^{\infty}$ abstracts to the pattern $((1, 0, 1, \mathrm{E}), \mathrm{RIGHT})$.

Across the $S$-step run, recorded configurations are grouped into a dataset keyed by their corresponding abstract pattern. We then rank these patterns from most to least common. This ranking allows us to select the top $K$ most frequent patterns for the subsequent automated conjecturing phase.

### 3.2 Automated Conjecturing of Transition Rules

Following pattern identification, the pipeline attempts to deduce the underlying mathematical rules governing the transitions between instances of the same pattern. For the top $K$ most frequent patterns, we extract a dataset of observed transitions from the initial $S$-step simulation. Each sample pairs successive configurations belonging to the target pattern, mapping initial block exponents to those at the next occurrence. For example, within the pattern $((1, 0, 1, \mathrm{E}), \mathrm{RIGHT})$ from Antihydra, an observed transition maps the configuration $0^{\infty}\ 1^2\,0\,1^8\ \mathrm{E}{>}\ 0^{\infty}$ to $0^{\infty}\ 1^4\,0\,1^{14}\ \mathrm{E}{>}\ 0^{\infty}$, corresponding to the exponent mapping $(2, 1, 8) \to (4, 1, 14)$.

We provide this empirical transition dataset to a Large Language Model (LLM), tasking it with generating an executable Python function that maps input exponent tuples deterministically to subsequent tuples.

Expressing conjectures as code grants immediate empirical evaluability. We evaluate candidate functions by executing them across the recorded transitions, measuring their prediction accuracy. For a pattern parameterized by three exponents, $(x, y, z)$, the LLM synthesizes a function modeling the cycle dynamics, as illustrated below for Antihydra. Here the middle exponent always equals 1, so that $(x, 1, z)$ corresponds to the configuration $A(x, z)$ of Section 2.2:

```python
def transition_rule(x: int, y: int, z: int) -> tuple[int, ...] | None:
    # Case: z is even (A(a, 2b) -> A(a+2, 3b+2))
    if z % 2 == 0:
        return (x + 2, 1, 3 * (z // 2) + 2)
    # Case: z is odd (A(a, 2b+1) -> A(a-1, 3b+3))
    else:
        return (x - 1, 1, 3 * (z // 2) + 3)
```

While this synthesized function provides an initial conjecture, it is often incomplete as it is strictly derived from the configurations observed during the initial simulation. Specifically, it overlooks rare edge cases absent from the initial simulation. For instance, if execution never reaches a state with $z$ odd and $x = 0$, the data will lack this boundary condition. Consequently, the naive rule above produces an invalid negative exponent ($-1$) rather than signaling termination, motivating an active refinement phase.

### 3.3 Interactive Refinement of Conjectures

Simply extending the initial simulation offers diminishing returns: as tape blocks expand, witnessing a single subsequent transition demands exponentially more raw Turing machine steps. Longer traces thus supply few additional examples.

To overcome this constraint, we equip the LLM with an interactive simulator to test conjectures on arbitrary synthetic configurations. When the model detects an edge case or requires data to validate a hypothesis, it queries this tool with custom exponent tuples. The simulator instantiates the exact hypothetical configuration; for example, given exponents $x = 0, y = 1, z = 9$, it constructs $0^\infty\, 1^0\, 0\, 1^9\, \mathrm{E}{>}\, 0^\infty$ and simulates execution until the machine re-encounters the pattern, terminates, or exceeds a step bound.

The model tests configurations regardless of blank-tape reachability, isolating missing branches absent from empirical traces. This capability proves vital for hypothesis refinement: continuing our previous example with Antihydra, by actively probing the case where $z$ is odd and $x = 0$, the agent can observe actual termination rather than invalid negative exponents.

This cycle of hypothesizing, targeted simulation, and code revision iterates up to $N$ times, methodically transforming incomplete guesses into precise transition rules.

Finally, the agent translates the validated code into a piecewise mathematical function over block exponents, algebraically accelerating multi-step sequences where possible. For Cryptids, these rules exhibit clear Collatz-like dynamics. For the running example, the final formal conjecture becomes:

1. $A(0, 2k+1) \to$ `halt` for $k \geq 0$.
2. $A(a, 2k) \to A(a+2, 3k+2)$ for $a \geq 0, k \geq 1$.
3. $A(a, 2k+1) \to A(a-1, 3k+3)$ for $a \geq 1, k \geq 0$.

### 3.4 Informal Proving

Before proceeding to mathematical verification, we apply a strict empirical filter. We evaluate candidate Python functions across all recorded transitions, combining the initial simulation trace with synthetic samples from interactive exploration. Only conjectures achieving perfect empirical accuracy (correctly predicting 100% of transitions) enter the informal proving phase.

Because conjectured rules decompose into distinct cases (e.g., modular parity or boundary conditions), the system isolates each rule and proves it independently by tracking abstract tape dynamics across multiple steps.

To generate informal proofs autonomously, we deploy the LLM in an iterative generator-evaluator loop. At each cycle, the model assumes three specialized roles:

1. **Proof Generator:** Drafts a step-by-step mathematical argument demonstrating that the input configuration deterministically evolves into the output configuration.
2. **Correctness Evaluator:** Audits mathematical validity, assigning a score $c \in \{0, 0.5, 1\}$:

- 1.0: Fully rigorous; all abstract execution steps are valid and clearly demonstrated.
- 0.5: Conceptually sound, but contains minor arithmetic errors or skips intermediate steps.
- 0.0: Contains invalid transitions, hallucinates rules, or exhibits major logical flaws.

3. **Format Evaluator:** Assesses clarity, structure, and exposition, assigning a score $f \in \{0, 0.5, 1\}$:
   - 1.0: Well-structured and adheres strictly to mathematical exposition standards.
   - 0.5: Legible, but certain arguments lack sufficient detail or contextual framing.
   - 0.0: Incoherent, excessively truncated, or poorly articulated.

If a candidate fails to achieve $c = 1$ and $f = 1$, the evaluators' critique and targeted diagnostic feedback are routed back to the Proof Generator for revision. This critique-and-refine cycle iterates up to $M$ times, terminating early once both criteria are fully met.

Upon receiving top scores from both evaluators, the argument is accepted as an approved informal proof. This workflow is repeated across all case-specific transitions before progressing to formal verification in Lean 4.

### 3.5 Formal Proving

The informal proving phase establishes strong confidence in the generated conjectures. However, to formally guarantee that the conjectured function captures the Turing machine's long-term behavior, the final stage of our pipeline verifies the results in Lean 4. For this, we adapt definitions and lemmas from the BusyLean repository (Yuen, 2025) which are necessary for reasoning about Turing machines in Lean, and provide them in-context to the agent. This allows the agent to reason about Collatz-like transition rules over abstract families of tape configurations (patterns). We also supply the agent with the informal proofs generated during the previous phase, along with the initial auto-formalized Lean statements about the various transition rules, as additional context. Together, these proofs and formal statements act as a scaffold, guiding the agent as it formally defines and verifies the Turing machine's long-term behavior.

**1. Auto-Formalization and Initialization:** The process begins by providing the agent with three sets of formalized statements: an equivalence statement in Lean between the Turing machine and the previously derived set of rules, a system closure statement, and the target halting or non-halting theorems. These are given as initial context to a Gemini-powered multi-persona agent tailored for Lean theorem proving, alongside the candidate machine's definitions and the informally verified closure proofs.

Our approach is fundamentally different from existing auto-formalization approaches explored in Lean. Existing auto-formalization approaches treat the informal math text as ground truth and attempt a one-to-one translation from English into a formal language. However, informal math often omits edge cases, ignores variable bounds, or relies on implicit domain knowledge. Lean's strict type theory checker does not tolerate such handwaving, and so a strict translation of slightly inaccurate informal text inevitably yields an unprovable formal theorem.

To overcome this, our agentic prover treats the initially seeded Lean statements not as rigid specifications, but as guidance. The agent is allowed to modify and refine the initially seeded Lean statements as it searches for a proof. Specifically, it can adjust rules to add missing bounds to parameters or introduce necessary new rules that were omitted from the initial system closure statement. This capability mirrors the workflow of an expert mathematician: attempting a proof, identifying a missing boundary condition, and subsequently refining the initial theorem statement so the proof succeeds. Furthermore, if the agent determines the existing rules are insufficiently comprehensive to define the system closure, it may introduce and prove additional transition rules, and integrate them into the closure statement, as long as these new additions do not trivialize the theorem by introducing false hypotheses.

**2. The Multi-Persona Architecture:** To translate, plan, prove, and verify the formal math statements, the system orchestrates four specialized persona roles:

**Problem Setter:** Formalizes the initially seeded rules as Lean statements. Since the initial conjecture may be wrong on edge cases, it may modify or add rules when the Writer or Critic reports an unprovable statement, testing candidates by simulation. Only the rules may change: the Lean definition of the machine and the start configuration (reached from the blank tape in a few steps, checked

by simulation) are fixed. The final system is valid if each rule is proven against the machine and the system is closed, i.e., the start configuration and every rule's output either halt or match some rule's input.

**Planner:** Decomposes the overall task of proving the various theorem statements into smaller lemmas, identifies gaps in existing proofs, and coordinates the workflow between the Writer and the Critic.

**Writer:** Constructs the formal proofs for the various theorem statements. It uses the informal proofs as guidance, translating them into Lean.

**Critic:** Acts as a verifier, ensuring that the proofs generated by the Writer are valid. The Critic evaluates feedback generated by the Writer during its proof attempts. This is a crucial step in the agent orchestration, as the Writer might be tackling an unprovable statement and passes this feedback to the Problem Setter. The Critic is responsible for granting final approval. It audits the Problem Setter's formalizations for accuracy and inspects the Writer's proofs to ensure that no unproven steps remain, no illegal axioms are used, and no reward hacks are exploited.

**3. Orchestration and Error Correction:** These personas operate over a shared workspace through a round-robin schedule, taking sequential turns to coordinate. Crucially, the auto-formalized statements are not static but are part of a bi-directional feedback loop. If the **Critic** or **Writer** identifies a flawed or unprovable rule, or if the Lean compiler throws an error, this feedback flows back to the **Problem Setter**. To diagnose and correct these errors, the agent is equipped with simulation tools to execute the Turing machine from a given pattern and empirically test the rules before the Problem Setter modifies the formal statements.

**4. Termination Criteria:** This iterative refinement loop continues until a final state is reached. The agent terminates successfully once the **Critic** confirms that all required Lean theorems compile without remaining unproven goals (`sorry`) or unauthorized axioms. Alternatively, it terminates unsuccessfully if a predefined time limit (72 hours) is reached.

# 4 Experiments and Results

## 4.1 Implementation

We implement our pipeline using Gemini 3.1 Pro across all agentic components, integrating custom Python simulation tools with the Lean 4 interactive theorem prover.

**Pattern Detection and Conjecturing.** For initial pattern detection, each Turing machine is simulated for $S = 5 \times 10^6$ steps, from which we extract the top $K = 100$ most frequent abstract patterns. Automated conjecturing and programmatic transition rule synthesis are powered by Gemini 3.1 Pro. During interactive refinement, the targeted simulation tool enforces a limit of $128,000$ steps per synthetic transition, and the agentic refinement loop executes for up to $N = 5$ turns.

**Informal and Formal Verification.** Synthesized rules must achieve a $100\%$ prediction accuracy across all empirical and synthetic transition data to pass the initial evaluation filter. The informal proving stage uses Gemini 3.1 Pro for all three roles in the generator-evaluator loop, which is capped at $M = 4$ iterations per rule. Finally, formal theorem proving is driven by an autonomous Gemini 3.1 Pro agent interacting directly with the Lean 4.28.1 environment.

## 4.2 Experiments

To evaluate our pipeline, we first executed the Pattern Detection and Automated Conjecturing stages across three distinct sets of Turing machines, restricting the system to a single conjecturing attempt per pattern. We deployed the system across the active frontier of unclassified machines: 1,094 $\mathrm{BB}(6)$ holdouts compiled by mxdys, the 65 remaining holdouts in $\mathrm{BB}(2,5)$, and the 4 remaining holdouts in $\mathrm{BB}(3,3)$.

This initial broad search identified 53 promising machines (51 of which belonged to the $\mathrm{BB}(6)$ domain) that each yielded two or more conjectures passing the strict 100% empirical accuracy threshold. To increase our chances of generating conjectures that could be formally proven, we isolated

these 53 machines and re-ran the conjecturing pipeline with higher-resolution settings, raising the allowance to 4 conjecturing attempts per pattern.

All conjectures from these high-resolution runs that maintained 100% empirical accuracy were advanced to the informal proving phase. Conjectures that successfully satisfied both the correctness and format evaluations were then submitted to the autonomous agent for formal theorem proving in Lean.

### 4.3 Results

Following this end-to-end verification, our pipeline produced formal proofs governing several complex machines. We manually validated all final Lean statements, including the machine definitions, start configurations, and closure theorems. All Lean proofs are provided in the supplementary material. Table 1 details these results across $\mathrm{BB}(6)$ and $\mathrm{BB}(2,5)$. We categorize outcomes by **Discovery type** to clarify their contribution relative to prior community findings:

- **New Discoveries:** Novel accelerations for 5 machines lacking prior documented community analyses.
- **Re-Discoveries:** Alternative or higher-level accelerations for 4 machines previously explored via unverified conjectures, such as informal Discord proposals.

Complete transition systems synthesized by our system appear in Appendix B.

## 5 Conclusion

In this work, we explored the Busy Beaver frontier as an open-ended testbed for automated mathematical discovery and formal verification. We introduced a pipeline that combines empirical pattern detection, LLM-guided conjecturing, and informal proving through a generator-evaluator loop to derive Collatz-like accelerations for Turing machines, expressed as systems of transition rules between parametrized tape configurations. These transition rules are then formally verified in Lean 4, establishing the correctness of each acceleration.

The empirical success of our pipeline was demonstrated by the discovery and formal verification of accelerations within the $\mathrm{BB}(6)$ and $\mathrm{BB}(2,5)$ domains. Our system successfully derived and formally verified novel Collatz-like accelerations for five machines in $\mathrm{BB}(6)$. Subject to community acceptance, we classify these machines as Cryptids in the sense of Definition 2.1. These results represent a step toward systematically classifying the remaining holdouts in the Busy Beaver search.

By design, the Busy Beaver domain is fundamentally open-ended due to its incomputability, making it a compelling testbed for open-ended AI research (Hughes et al., 2024). However, we acknowledge that our current approach is not entirely open-ended. Specifically, our system currently accelerates machines exhibiting Collatz-like functions—structures already studied by mathematicians and whose long-term behavior can be exceptionally difficult to resolve. A more ambitious version of this work would relax these assumptions, expanding beyond Collatz-like dynamics to other unsolved mathematical problems. The ultimate goal would be a fully autonomous system capable of discovering entirely novel classes of open problems and mapping Turing machines to them.

While the Busy Beaver challenge acts as a generator of mathematical questions—given that important open problems can be encoded as Turing machines whose halting status must ultimately be resolved—the broader mathematical significance of the questions arising from its exploration is not guaranteed. For instance, although progress on specific Cryptids could theoretically yield insights into the Collatz conjecture, such outcomes remain speculative and cannot be guaranteed. Regardless of these domain-specific caveats, our architecture serves as a blueprint for broader AI-driven mathematical discovery. By equipping large language models with interactive computational tools, generator-evaluator critique loops, and iterative formal proving environments, we add to the growing body of work on autonomous systems capable of tackling increasingly profound challenges in theoretical computer science.

### AI USE STATEMENT

In this work, we used generative AI tools (specifically Gemini 3.1 Pro and Antigravity) for deriving formal and informal theoretical proofs, writing and refactoring portions of experimental code, and generating synthetic evaluation data. We have not used generative AI tools for core research ideation, formulating original research hypotheses, or drafting the core substantive manuscript text.

Additionally, we used generative AI tools for literature search, summarization of related works, and linguistic proofreading and copy-editing.

We have reviewed all AI-assisted work. All derived mathematical formulations and proofs were manually checked line-by-line and rigorously verified by the authors. The codebase was primarily author-written; all AI-assisted implementation was manually code-reviewed and verified against experimental baselines. All literature citations and factual claims were independently cross-referenced with original source publications. We take full responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

### ETHICS STATEMENT

This work complies with the ICLR Code of Ethics. Our research focuses on foundational methodology and does not involve human subjects, crowdsourced user studies, or personally identifiable information (PII). We do not collect or release sensitive or copyrighted datasets, and our methodology does not present immediate dual-use, fairness, or safety concerns. Consequently, no specialized institutional oversight or ethical clearance was required.

### REPRODUCIBILITY STATEMENT

To support reproducibility, we have detailed our formal theoretical assumptions and provided complete step-by-step mathematical proofs in the supplementary material. All primary experimental results rely on algorithms documented within the text, and the implementation code, synthetic evaluation generators, and runtime scripts will be made publicly available via an open-source GitHub repository upon publication.

## REFERENCES

Scott Aaronson. The busy beaver frontier. *ACM SIGACT News*, 51(3):32–54, 2020.

Noga Alon, Thomas F Bloom, W Timothy Gowers, Daniel Litt, Will Sawin, Arul Shankar, Jacob Tsimerman, Victor Wang, and Melanie Matchett Wood. Remarks on the disproof of the unit distance conjecture. *arXiv preprint arXiv:2605.20695*, 2026.

Justin Blanchard, Daniel Briggs, Konrad Deka, Nathan Fenner, Yannick Forster, Georgi Georgiev, Matthew L House, Maja Kadziołka, Pavel Kropitz, Shawn Ligocki, et al. Determination of the fifth busy beaver value. In *Proceedings of the 58th Annual ACM Symposium on Theory of Computing*, pp. 565–574, 2026.

Gregory Chaitin. *Proving Darwin: making biology mathematical*. Vintage, 2013.

Tony Feng, Trieu H Trinh, Garrett Bingham, Dawsen Hwang, Yuri Chervonyi, Junehyuk Jung, Joonkyung Lee, Carlo Pagano, Sang-hyun Kim, Federico Pasqualotto, et al. Towards autonomous mathematics research. *arXiv preprint arXiv:2602.10177*, 2026.

Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Petar Sirkovic, Artiom Myaskovsky, Grzegorz Glowaty, Felix Weissenberger, Alessio Orlandi, Dan Popovici, Anil Palepu, Keran Rong, Ryutaro Tanno, Khaled Saab, Fan Zhang, Jacob Blum, Andrew Carroll, Kavita Kulkarni, Nenad Tomašev, Dina Zverinski, et al. Accelerating scientific discovery with co-scientist. *Nature*, 2026. doi: 10.1038/s41586-026-10644-y. URL https://www.nature.com/articles/s41586-026-10644-y.

William Timothy Gowers. The two cultures of mathematics. *Mathematics: frontiers and perspectives*, 65:65, 2000.

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651(8106):607–613, 2026.

Edward Hughes, Michael Dennis, Jack Parker-Holder, Feryal Behbahani, Aditi Mavalankar, Yuge Shi, Tom Schaul, and Tim Rocktaschel. Open-endedness is essential for artificial superhuman intelligence. *arXiv preprint arXiv:2406.04268*, 2024.

Po-Nien Kung, Linfeng Song, Dawsen Hwang, Jinsung Yoon, Chun-Liang Li, Simone Severini, Mirek Olšák, Edward Lockhart, Quoc V Le, Burak Gokturk, et al. Leap: Supercharging llms for formal mathematics with agentic frameworks. *arXiv preprint arXiv:2606.03303*, 2026.

Junqi Liu, Zihao Zhou, Zekai Zhu, Marco Dos Santos, Weikun He, jiawei liu, Yunzhou Xie, Junqiao Zhao, Qiufeng Wang, Lihong Zhi, Jia LI, and Wenda Li. Numina-lean-agent: An open and general agentic reasoning system for formal mathematics. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=0bTEd4LpQr.

Chris Lu, Cong Lu, Robert Tjarko Lange, Yutaro Yamada, Shengran Hu, Jakob Foerster, David Ha, and Jeff Clune. Towards end-to-end automation of ai research. *Nature*, 651(8107):914–919, 2026.

Pascal Michel. The busy beaver competition: a historical survey. *arXiv preprint arXiv:0906.3749*, 2009.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI. Finite time blowup for navier–stokes. Technical report, OpenAI, 2026. URL https://cdn.openai.com/pdf/32d9f210-8b73-45e0-91bc-82a30aef8a9a/navier-stokes.pdf.

Tibor Rado. On non-computable functions. *Bell System Technical Journal*, 41(3):877–884, 1962.

Anirudha Ramesh and Shreyas Pimpalgaonkar. Conjecturebench: a curated collection of open mathematics problems, 2026. URL https://github.com/bespokelabsai/conjecture-bench. Bespoke Labs.

Zhihong Shao, Yuxiang Luo, Chengda Lu, ZZ Ren, Jiewen Hu, Tian Ye, Zhibin Gou, Shirong Ma, and Xiaokang Zhang. Deepseekmath-v2: Towards self-verifiable mathematical reasoning. *arXiv preprint arXiv:2511.22570*, 2025.

Terence Tao. Mathematics in the age of ai, 2026. URL https://arxiv.org/abs/2608.16753.

George Tsoukalas, Anton Kovsharov, Sergey Shirobokov, Anja Surina, Moritz Firsching, Gergely Bérczi, Francisco J. R. Ruiz, Arun Suggala, Adam Zsolt Wagner, Eric Wieser, Lei Yu, Aja Huang, Miklós Z. Horváth, Andrew Ferraiuolo, Henryk Michalewski, Edward Lockhart, Codrut Grosu, Thomas Hubert, Matej Balog, Pushmeet Kohli, and Swarat Chaudhuri. Advancing Mathematics Research with AI-Driven Formal Proof Search, 2026. URL https://arxiv.org/abs/2605.22763.

Jason Yuen. BusyLean. https://github.com/int-y1/proofs/tree/master/BusyLean, 2025. GitHub repository.

Daniel Zheng, Ingrid von Glehn, Yori Zwols, Iuliya Beloshapka, Lars Buesing, Daniel M Roy, Martin Wattenberg, Bogdan Georgiev, Tatiana Schmidt, Andrew Cowie, et al. Ai co-mathematician: Accelerating mathematicians with agentic ai. *arXiv preprint arXiv:2605.06651*, 2026.

Tianshi Zheng, Zheye Deng, Hong Ting Tsang, Weiqi Wang, Jiaxin Bai, Zihao Wang, and Yangqiu Song. From automation to autonomy: A survey on large language models in scientific discovery. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 17744–17761, 2025.

# A Related Work

Recent advancements in neural theorem proving have demonstrated remarkable capabilities in both formal and informal mathematical reasoning. Over the past two years, state-of-the-art systems have achieved historic milestones, including reaching silver and gold medal levels at the International Mathematical Olympiad and attaining near-perfect scores on the Putnam exam (Hubert et al., 2026; Shao et al., 2025; Liu et al., 2026). Pushing beyond these standardized benchmarks, frontier models and systems have successfully resolved open mathematical questions (Feng et al., 2026; Tsoukalas et al., 2026; Alon et al., 2026). While these systems excel at navigating complex search spaces to find proofs, they were not designed to autonomously generate novel mathematical conjectures or formulate new theories. Beyond problem solving, frameworks such as the AI Co-Mathematician (Zheng et al., 2026) leverage AI agents for open-ended research through ideation, literature search, computational exploration, theorem proving, and theory building. However, they function primarily as collaborative assistants that rely on human intuition to define the mathematical objects of interest and guide the direction of exploration. Taking a fully autonomous approach, evolutionary systems like AlphaEvolve (Novikov et al., 2025) explore open-ended spaces by mutating code to discover novel algorithmic heuristics. However, their strict reliance on a climbable reward signal fundamentally restricts their application to general mathematics, where such smooth, quantifiable optimization gradients rarely exist.

Recently, several AI systems have emerged to automate the scientific discovery process, spanning from hypothesis generation (Gottweis et al., 2026; Zheng et al., 2026) to experimental analysis (Zheng et al., 2025). Notably, Sakana AI (Lu et al., 2026) introduced an end-to-end automated scientist for machine learning that autonomously conducts research—from ideation and coding to manuscript writing and peer review—ultimately producing a workshop-accepted paper. While such frameworks are theoretically generalizable, empirical domains like machine learning are inherently easier to evaluate than mathematics, because in the latter establishing a conjecture requires a mathematical proof, although experiments can help test and refine it. To address this challenge, newer frameworks divide mathematical discovery into distinct phases: conjecture generation and proof writing. As an example, ConjectureBench (Ramesh & Pimpalgaonkar, 2026) isolates the conjecturing phase, producing statements that automated theorem provers can then attempt to verify. Building on this, AI systems like LEAP (Kung et al., 2026) can autonomously attempt to prove theorems by pairing an ideation agent that drafts informal proofs with a formalization agent that translates them into verified Lean 4 code, marking significant progress toward end-to-end automated discovery in mathematics.

Applying these powerful AI-driven methodologies provides a new approach to classical challenges in computability theory, most notably the Busy Beaver problem originally introduced by Radó (Rado, 1962). Historically, evaluating these theoretical limits has relied on exhaustive manual heuristics to filter out non-halting Turing machines, leaving behind complex "holdouts" whose behaviors are intimately tied to deep mathematical conjectures (Michel, 2009; Aaronson, 2020). The implications of these limits reveal the uncomputable, open-ended nature of algorithmic generation (Chaitin, 2013). Despite this immense computational complexity, the recent monumental proof resolving the fifth Busy Beaver value was achieved entirely by a collaborative community effort without the use of artificial intelligence (Blanchard et al., 2026). More recently, in other models of computation, AI is already making direct mathematical contributions; most notably, the large language model Claude Opus 4.6 was recently used to successfully generate proofs of non-halting for hundreds of complex FRACTRAN holdout machines, entirely resolving the remaining $\mathrm{BB}_f(21)$ cases and proving all but the 'Fenrir-family' Cryptids in the $\mathrm{BB}_f(22)$ domain (Wiki).

# B Accelerations Discovered by Our Pipeline

This section presents the accelerations discovered and formally verified by our pipeline. These comprise novel accelerations for five previously unanalyzed machines in $\mathrm{BB}(6)$, together with newly derived higher-level or alternative acceleration systems for four previously studied machines: one in $\mathrm{BB}(2,5)$ and three in $\mathrm{BB}(6)$. The corresponding Lean proofs are provided in the supplementary material.

### B.1 NEW DISCOVERY: 1RB1LD_1RC0RA_1LA1RE_0LA0LB_0RB1RF_---1RD

1RB1LD_1RC0RA_1LA1RE_0LA0LB_0RB1RF_---1RD is a holdout in BB(6) for which no acceleration had previously been documented by the community. Our pipeline discovered and formally verified the following Collatz-like acceleration for this machine:

```
Consider:
A(a, b, c, d) = 0^inf 1^a 10^b A> 01^c 1^d 0^inf

A(a, b, c, d) -> A(a, b+1, c-1, d) if c > 0
A(a, b, 0, 0) -> A(a+1, 0, 0, 2b+2)
A(a, b, 0, d) -> A(a-3, b+2, 0, d-1) if d > 0 and a >= 3
A(2, b, 0, d) -> A(0, 0, b+1, d-1) if d > 0
A(1, b, 0, d) -> A(1, 1, b+1, d-1) if d > 0 and b > 0
A(1, 0, 0, d) -> A(0, 0, 0, d+2) if d > 0
A(0, 0, 0, 1) -> A(0, 0, 2, 5)
A(0, b, 0, 1) -> A(0, 0, 2, 2b+1) if b > 0
A(0, b, 0, d) -> A(3d-4, 1, 2, 2b-2d+3) if d >= 2 and b >= d-1
A(0, b, 0, b+3) -> A(3b+2, 0, 0, 6)
A(0, b, 0, b+4) -> A(3b+4, 0, 0, 4)
A(0, b, 0, d) -> A(3b+3, 2, 0, d-b-5) if d >= b+5
A(0, b, 0, b+2) -> Halt

Start: A(0, 0, 0, 0), already present on the initial blank tape
```

### B.2 NEW DISCOVERY: 1RB0LD_1LC1RA_1RA1LC_1LE1LA_0RF1LD_---1RE

1RB0LD_1LC1RA_1RA1LC_1LE1LA_0RF1LD_---1RE is a holdout in BB(6) for which no acceleration had previously been documented by the community. Our pipeline discovered and formally verified the following Collatz-like acceleration for this machine:

```
Consider:
C(a, g, h) = 0^inf 01^a 1^g A> 1^h 0^inf

Let:
S(j) = 2^(j+1) - j - 1

And:
OddTarget(a,2k+1,h):
  choose j such that S(j) <= a < S(j+1)
  if a = S(j):
    C(2^(j+1)+k, 2, h-2)
  otherwise, with a' = S(j+1)-1-a:
    C(a', 2*(a+j+2-a')+2k, h-2)

C(0, g, 0) -> C(0,1,g+2)
C(a, g, 0) -> C(a-1,1,g+3) if a >= 1
C(0, g, h) -> C(g/2+1,2,h-2) if h >= 3 and g >= 2 and g  is even
C(a, g, h) -> C(a-1, g+4, h-2) if h >= 3, g is even, a >= 1
C(0, g, h) -> C(0, g+3, h-2) if h >= 3 and g is odd
C(a, 2k+1, h) -> OddTarget(a,2k+1,h) if h >= 3 and a >= 1
C(0, g, 2) -> Halt if g is even
C(0, g, 2) -> C(0, g+3, 0) if g is odd
C(a, g, 2) -> C(a-1, g+4, 0) if a >= 1 and g is even
C(a, 2k, 1) -> C(a, 2k-2, 2) if k > 1
C(0, 2, 1) -> Halt
C(a, 2, 1) -> C(a-1, 4, 0) if a >= 1

Start: C(0, 1, 2), reached after 4 raw TM steps
```

### B.3 NEW DISCOVERY: 1RB1RD_0LC1RA_0RA1LC_1RA1LE_1RF0LD_---0RE

1RB1RD_0LC1RA_0RA1LC_1RA1LE_1RF0LD_---0RE is a holdout in $\mathrm{BB}(6)$ for which no acceleration had previously been documented by the community. Our pipeline discovered and formally verified the following Collatz-like acceleration for this machine:

```
Consider:
C(a, c) = 0^inf 1^a A> 0 1^c 0^inf

T(0, k, n) = (2k, n+1)
T(x+1, k, n) =
  T(x, k+2, n-2) if n >= 2
  T(x, 1, 2k+3) if n = 1
  T(x, 1, 2k+1) if n = 0

C(0, 0) -> C(2, 0)
C(a, 0) -> C(1, a) if a >= 1
C(a, 1) -> C(1, a+2) if a >= 1
C(a, 2) -> C(1, a+4) if a >= 1
C(4m+1, 3) -> Halt
C(4m+2, c) -> C(4m+5, c-2) if c >= 3
C(4m+1, 4) -> C(T(2m+2, 0, 0))
C(4m+1, c) -> C(T(2m+2, 1, c-5)) if c >= 5
C(4m, c) -> C(4m+3, c-2) if m >= 1 and c >= 3
C(4m+3, 3) -> Halt

Start: C(0,0), already present on the initial blank tape
```

### B.4 NEW DISCOVERY: 1RB1RB_1LC0RE_---1LD_0RA1LB_1RA1LF_0LE1LE

1RB1RB_1LC0RE_---1LD_0RA1LB_1RA1LF_0LE1LE is a holdout in $\mathrm{BB}(6)$ for which no acceleration had previously been documented by the community. Our pipeline discovered and formally verified the following Collatz-like acceleration for this machine:

```
Consider:
C(a, c, f) = 0^inf 1^a 0 1^c 0 1 <D 1^f 0^inf

C(2k, 2m, f) -> C(2, 2m-2k+2, f+6k-6) if 2 <= k < m, f >= 1
C(2k, 2m, f) -> C(2k-2m+2, 2, f+6m-6) if 2 <= m < k, f >= 1
C(2k, 2k, f) -> C(2, 2, f+6k-6) if k >= 2, f >= 1
C(2k, 2, 2j)   -> C(2k-4, 2j+8, 2) if k >= 3, j >= 1
C(2k, 2, 2j+1) -> C(2k-4, 2j+8, 5) if k >= 3, j >= 0
C(4, 2, 2j)    -> C(2j+4, 6, 5) if j >= 1
C(4, 2, 2j+1)  -> C(2j+8, 2, 2) if j >= 0
C(2, 2m, 2j)   -> C(2m-6, 2j+10, 5) if m >= 4, j >= 1
C(2, 2m, 2j+1) -> C(2m-6, 2j+12, 2) if m >= 4, j >= 0
C(2, 6, 2j)    -> C(2j+10, 2, 2) if j >= 1
C(2, 6, 2j+1)  -> C(2j+8, 6, 5) if j >= 0
C(2, 4, f) -> Halt if f >= 1
C(2, 2, f) -> Halt if f >= 1

Start: C(6, 2, 2), reached after 76 raw TM steps
```

### B.5 NEW DISCOVERY: 1RB0LF_1LC1RA_0RE0RD_---1LE_1LF1RC_1LC1LA

1RB0LF_1LC1RA_0RE0RD_---1LE_1LF1RC_1LC1LA is a holdout in $\mathrm{BB}(6)$ for which no acceleration had previously been documented by the community. Our pipeline discovered and formally verified the following Collatz-like acceleration for this machine:

```
Consider:
C(a, b) = 0^inf E> 10^a 1^b 0^inf

C(a, b) -> C(2a+2, b-2a-2) if b > 2a+2
C(a, 1) -> C(2, 2a+2)
```

```
C(a, 2)  -> Halt
C(a, 2k) -> C((6a+2k+2)/4, (6a+2k+2)%4+1) if a >= 1 and 2 <= k <= a+1
C(a, 2k+1) -> C((6a+2k+6)/4, (6a+2k+6)%4+1) if a >= 1 and 1 <= k <= a

Start: C(0,3), reached after 7 raw TM steps
```

### B.6 RE-DISCOVERY: 1RB3LA1LA1RA1RA_2LB2RA---4RB1LB

1RB3LA1LA1RA1RA_2LB2RA---4RB1LB is a holdout in $\mathrm{BB}(2, 5)$ for which an acceleration had previously been documented by mxdys on the Wiki. Our pipeline discovered and formally verified the following Collatz-like acceleration for this machine:

```
Consider:
A(a, b, c) = 0^inf 1 4^a <A 3^b 1^c 0^inf

Let:
B_k = 2^(k+1) - k - 7
S_k = 2^(k+1) - 2k - 10

For a given b, let k >= 3 be the smallest integer such that b <= B_k.

A(1, b, 2) -> A(b+2, 2, 2)
A(a, b, 2) -> A(a-1, b+2, 1) if a > 1
A(1, b, 1) -> A(2b-S_k, 2, 2) if b = B_k
A(1, b, 1) -> A(B_k-b, 2b-S_k, 1) if b < B_k and 2b > S_k
A(1, b, 1) -> Halt if b < B_k and 2b = S_k
A(2, b, 1) -> A(b+2, 4, 2)
A(a, b, 1) -> A(a-2, b+3, 2) if a > 2

Start: A(2, 4, 2), reached after 49 raw TM steps
```

### B.7 RE-DISCOVERY: 1RB0RB_1LB0LC_1RD1LC_1RE0RA_0RF1RA_---1LB

1RB0RB_1LB0LC_1RD1LC_1RE0RA_0RF1RA_---1LB is a holdout in $\mathrm{BB}(6)$ for which an acceleration had previously been documented by Sheep on Discord. Our pipeline discovered and formally verified the following Collatz-like acceleration for this machine:

```
Consider:
A(a,b) = 0^inf 1^a <B 1^b 0^inf

A(1, 3k) -> Halt if k >= 1
A(1, 3k+1) -> A(3k+4, 1)
A(1, 3k+2) -> A(3k+3, 2)
A(3k, b) -> A(k, b+2k+1) if k>=1 and b >= 1
A(3k+1, b) -> A(4k+b+3, 1) if k>=1 and b >= 1
A(3k+2, b) -> A(4k+b+5, 1) if b >= 1

Start: A(1, 1), reached after 2 raw TM steps
```

### B.8 RE-DISCOVERY: 1RB---_0RC0RD_1LD1RB_0LE0LC_1RA0LF_1LD1LE

1RB---_0RC0RD_1LD1RB_0LE0LC_1RA0LF_1LD1LE is a holdout in $\mathrm{BB}(6)$ for which an acceleration had previously been documented by Shawn Ligocki on the Wiki. Our pipeline discovered and formally verified the following Collatz-like acceleration for this machine:

```
Consider:
A(a, c, d) = 0^inf 1^a 0 <E 0^c 1^d 0^inf

A(2k, 1, 1) -> Halt
A(2k+1, 1, 1) -> A(2k+6, 2, 1)
A(a, 1, d) -> A(4a+8, 1, d-3a-5) if d > 3a+5
A(a, 1, 3a+5) -> A(4a+6, 1, 1)
A(a, 1, 3k) -> A(4k+2, 2, a+2-k) if k >= 1 and 3k < 3a+5
A(a, 1, 3k+1) -> A(4k+4, 2, a+2-k) if k >= 1 and 3k+1 < 3a+5
```

```
A(a, 1, 3k+2) -> A(4k+4, 2, a+1-k) if 3k+2 < 3a+5
A(a, 2, d) -> A(4a+10, 1, d-3a-6) if d > 3a+6
A(a, 2, 3a+6) -> A(4a+8, 1, 1)
A(a, 2, 3k) -> A(4k+2, 2, a+2-k) if k >= 1 and 3k < 3a+6
A(a, 2, 3k+1) -> A(4k+4, 2, a+2-k) if 3k+1 < 3a+6
A(a, 2, 3k+2) -> A(4k+6, 2, a+2-k) if 3k+2 < 3a+6

Start: A(2, 2, 1), reached after 11 raw TM steps
```

### B.9 RE-DISCOVERY: 1RB1RF_0LC0RF_1RD1LC_---0LE_0RC1LF_1RA0LE

1RB1RF_0LC0RF_1RD1LC_---0LE_0RC1LF_1RA0LE is a holdout in $\mathrm{BB}(6)$ for which an acceleration had previously been documented by Shawn Ligocki on Discord. Our pipeline discovered and formally verified the following Collatz-like acceleration for this machine:

```
Consider:
A(a, b) = 0^inf 1^a A> 1^b 0^inf

A(2k, 1) -> A(3, 2k+2) if k >= 0
A(2k, 2) -> A(2k+1, 1) if k >= 0
A(2k, b) -> A(2k+3, b-2) if k >= 0 and b >= 3
A(2k+1, 2m) -> A(4m+3, 2k-2m+1) if k >= m >= 0
A(2k+1, 2m+1) -> A(4m+3, 2k-2m+3) if k >= m >= 0
A(2k+1, 2k+2) -> A(4k+1, 1) if k >= 0
A(2k+1, 2k+3) -> A(4k+7, 1) if k >= 0
A(2k+1, 2k+4) -> Halt if k >= 0
A(2k+1, 2k+5) -> A(4k+5, 1) if k >= 0
A(2k+1, b) -> A(4k+7, b-2k-5) if k >= 0 and b >= 2k+6

Start: A(1, 1), reached after 7 raw TM steps
```

## C TAPE COMPRESSION ALGORITHM

In this section, we present the algorithm we use to convert raw Turing machine configurations into a compact representation based on repeated tape symbols. The algorithm takes as input a Turing machine configuration, consisting in the head, its position, and the Turing machine tape. It detects repeated blocks (sequences of tape symbols) and replaces consecutive repetitions with the block and its repetition count, while preserving the head marker and infinite blank regions.

For example, the raw Turing machine configuration, written in directed head notation:

$$0^{\infty}\,01010101\,\mathrm{E}{>}\,1111111\,0^{\infty}$$

is compressed into:

$$0^{\infty}\,01^{4}\,\mathrm{E}{>}\,1^{7}\,0^{\infty}$$

where $0^{\infty}$ denotes an infinite sequence of blank symbols and E$>$ indicates that the machine is in state E with the head reading the symbol immediately to its right. The compressed representation is exact: expanding each block according to its repetition count recovers the original configuration.

We compress the finite sequences of symbols on either side of the head independently, keeping the head marker and infinite blank regions unchanged. Algorithm 2 performs this scan by alternating between two steps: identifying a repeating block and adjusting its boundary. Each side is scanned from its outer boundary toward the head. At a given position, we identify repeating sequences of symbols starting there and use a simplicity score to choose among them. We then check whether taking one fewer copy of the chosen block would allow a longer repetition to start at the resulting boundary. We append the chosen block and repetition count to the output and continue with the remaining symbols. The following two steps describe this procedure for a sequence written in its scanning order.

The first step, implemented in Algorithm 1, identifies the next block. Let $s$ denote the sequence of symbols still to be processed. For each candidate block $b$ starting at the beginning of $s$, we count its maximum number $n$ of consecutive copies. Among candidates with $n \geq 2$, we maximize the

simplicity score

$$\text{score}(b, n) = \frac{n}{|b|},$$

where $|b|$ is the number of symbols in $b$. This favors short, frequently repeated blocks when several decompositions are possible: $01010101 = 01^4 = 0101^2$, but the respective scores are 2 and $1/2$. Ties favor shorter blocks. If no candidate repeats, we select the first symbol with count 1. Candidate lengths range from 1 to $\lfloor \min(|s|, K)/2 \rfloor$, where $K$ is a search parameter (default 16, allowing blocks of length at most 8). This step returns the candidate block $b$ and its repetition count $n$ to the main procedure.

**Algorithm 1** Block selection using the simplicity score

**Require:** Nonempty sequence of symbols $s$; integer $K \geq 2$
**Ensure:** Block and repetition count $(b, n)$

**function** BESTBLOCK($s, K$)
  bestBlock $\leftarrow (s[0], 1)$; bestScore $\leftarrow 0$
  **for** blockLength $= 1, \ldots, \lfloor \min(|s|, K)/2 \rfloor$ **do**
    $b \leftarrow s[0 : \text{blockLength}]$; $n \leftarrow 1$
    **while** $(n+1)$blockLength $\leq |s|$ **and** $s[n\text{blockLength} : (n+1)\text{blockLength}] = b$ **do**
      $n \leftarrow n + 1$
    **end while**
    **if** $n \geq 2$ **and** score$(b, n) >$ bestScore **then**
      bestBlock $\leftarrow (b, n)$
      bestScore $\leftarrow$ score$(b, n)$
    **end if**
  **end for**
  **return** bestBlock
**end function**

**Algorithm 2** Sequence compression with boundary adjustment

**Require:** Finite sequence of symbols $s$; integer $K \geq 2$
**Ensure:** List compressed of blocks and repetition counts

compressed $\leftarrow [\,]$
**while** $|s| > 0$ **do**
  $(b, n) \leftarrow$ BESTBLOCK($s, K$)
  fullLength $\leftarrow n|b|$
  **if** $n \geq 2$ **and** fullLength $< |s|$ **then**
    shortLength $\leftarrow$ fullLength $- |b|$
    $(b_{\text{next}}, n_{\text{next}}) \leftarrow$ BESTBLOCK($s$[fullLength :], $K$)
    fullCoverage $\leftarrow$ fullLength $+ n_{\text{next}}|b_{\text{next}}|$
    $(b_{\text{next}}, n_{\text{next}}) \leftarrow$ BESTBLOCK($s$[shortLength :], $K$)
    shortCoverage $\leftarrow$ shortLength $+ n_{\text{next}}|b_{\text{next}}|$
    **if** shortCoverage $>$ fullCoverage **then**
      $n \leftarrow n - 1$
    **end if**
  **end if**
  Append $(b, n)$ to compressed
  $s \leftarrow s[n|b| :]$
**end while**
**return** compressed

Before committing to this candidate, the main procedure checks whether its boundary should be adjusted. Taking all $n$ copies can consume the start of another repetition, so we compare keeping $n$ copies with keeping $n - 1$. For each choice, we call Algorithm 1 on the remaining symbols and measure the total number of symbols covered by the current and next blocks. We take one fewer copy only if this strictly increases that total. This check is skipped when $n = 1$ or the selected block

reaches the end of the sequence. The next block is examined only to guide this decision, it is not yet added to the output.

For example, on input $1111110101010$, block selection gives $1^6$. The next block would be $01^3$, covering $6 + 6 = 12$ symbols in total and leaving a final $0$:

$$\underbrace{1^6\,01^3}_{12\text{ symbols}}\;0$$

Taking only five initial 1s instead exposes $10101010 = 10^4$, giving a total coverage of $5 + 8 = 13$:

$$\underbrace{1^5\,10^4}_{13\text{ symbols}}\;.$$

After this adjustment, we append the current block and its final repetition count to the output, remove the corresponding symbols from the beginning of $s$, and repeat both steps on the remaining sequence. The procedure terminates when $s$ is empty. Applying it to both sides of the head and restoring their original orientation yields the compressed configuration.