# THE ZERO PARADOX

# ZP-K: Computational Grounding of Self-Reference

*Initial: April 2026 | Current: 1.38, November 2026*

---

This document establishes the computational grounding of the Zero Paradox's central self-reference structure. On the framework's reading, ⊥ in the computational instantiation is not a state of a Turing machine: it is the ground state of a universal Turing machine — the state from which no external executor is required. Kleene's second recursion theorem supplies a fixed point of the self-application operator. The reading of that fixed point as ⊥'s computational counterpart splits in two, and no field of the KleeneStructure class asserts either half. Its identity half is retired: a code and ⊥ of a lattice are compared through the Quine-atom role (Motion.SelfOnlyAt, a point that steps to itself and to nothing else), which under the eval step no single code occupies (evalStepMotion_not_selfOnlyAt) and, for a computable map respecting equal output, a class of codes up to equal output does (codes_vs_classes). Its running half, that ⊥ actually executes, is the occurrence commitment.

The central result is a three-way equivalence within this framework, in the precise sense defined by Remark R-K.0: the structural roles of ⊥ of a lattice carrying AFAStructure — the Quine-atom role (CC-2, read in the lattice's AFA structure, where ⊥ fills it by the class fields bot_self_mem and quine_unique; the set-theoretic fact is Aczel 1988, p. 19, where AFA allows at most one solution of x = {x}; see ZP-J; in Mathlib's ZFSet no set occupies it, zfset_quine_role_unoccupied), bottom element (order-theoretic), and join identity (algebraic) — are PROVED to coincide, while the fourth role, Kleene fixed point (computational), is required as a typeclass field rather than derived. (See Remark R-K.0 for the precise scope: equivalences (1)–(3) are derived via T-EXEC; equivalence (4) is combined by KleeneStructure's typeclass requirement, not derived independently.)

---

## Section I: The Kleene Fixed Point

### I. Kleene's Second Recursion Theorem

Kleene's second recursion theorem is the computational fixed-point theorem. In the form Mathlib states it, for any partially computable F sending each program code to a partial function, some code c computes exactly F(c) (kleene_fixed_point_exists). Rogers' fixed-point theorem is the form for a total computable transformation g of codes: some code c computes the same function as g(c) (roger_fixed_point_exists). Applied to the self-application map of II below, the recursion theorem gives a program whose output at any input n equals its output at its own Gödel number plus n — a periodicity fixed point.

In Lean 4, this is formalized in Mathlib's computability library. For any partially computable transformation f of codes, there exists a code c such that eval c = f c. The theorem supplies a code without distinguishing one. Section IV tabulates the measured axiom footprints.

**Kleene's Second Recursion Theorem (Mathlib)**

For any partially computable f : Code → ℕ →. ℕ, there exists c : Code such that eval c = f c.

Applied to the selfApply transformation, this yields a code c satisfying eval c n = eval c (encode c + n) for all n — a periodicity condition with period c's own Gödel number, though not necessarily the least. For every such f the fixed-point codes form an infinite set (fixed_points_infinite, Kleene.lean § VIII).

## II. The Self-Application Map

The self-application map sends each code c to the partial function that runs c on c's own Gödel number plus an offset. A fixed point of self-application satisfies a periodicity condition: eval c n = eval c (encode c + n) for all n, with period equal to c's own Gödel number, though not necessarily the least. The periodicity condition is met by codes that ignore their input: every constant code satisfies it, and infinite_quine_family witnesses an unbounded family of fixed points with exactly those codes. So this predicate alone does not pin self-reference; Kleene.lean § I states what that does and does not imply about codes generally, and III below gives the non-degenerate case.

**Definition: selfApply and IsComputationalQuine (Kleene.lean § I)**

selfApply : Code → ℕ →. ℕ := fun c n ↦ eval c (encode c + n)

A code c is a computational Quine if eval c = selfApply c, i.e.:

∀ n, eval c n = eval c (encode c + n)

This is a periodicity condition: c's output at n equals its output at encode(c) + n. The encoding plays the role of the "address" of the program — c's behavior at n is the same as c's behavior at its own address plus n.

selfApply_partrec: selfApply is partially computable.

Proof: via Mathlib computability primitives — eval composed with encode and nat_add. Lean purity: standard foundational axioms only. ✓

**Theorem: computational_quine_exists**

There exists a computational Quine: ∃ c : Code, IsComputationalQuine c.

Proof: immediate from kleene_fixed_point_exists applied to selfApply, using selfApply_partrec.

Lean purity: standard foundational axioms only. ✓

Note on uniqueness: the Quine atom of an AFAStructure lattice is unique by the class field quine_unique (T-EXEC, ZP-J). Codes meeting IsComputationalQuine are not unique: the constant codes alone give infinitely many (infinite_quine_family). Self-printing codes are unique at the level of behaviour (selfprints_behaviour_injective, III below). That the lattice's Quine atom is its ⊥, and the only one, flows from ZP-J T-EXEC, not from either computational predicate.

## III. Non-degenerate Self-Reference

A sharper predicate is not met by constants. A code is self-printing (SelfPrints) when, on every input of channel 0, it returns its own Gödel number, and universal (Universal) when, on channel e + 1, it computes what the code numbered e computes. A code that is both exists (selfref_universal_exists), and infinitely many do (selfref_universal_infinite). No constant code is universal (the control example after selfref_universal_infinite in Kleene.lean § VIII), and each half is met by some code without the other: universal_not_selfprints, and Code.zero, which is self-printing and not universal. These are statements about the partial function a code computes; they state that such codes exist, and none says that a code is run.

Abundance holds for every transformation: for any partially computable F, the codes c with eval c = F(c) form an infinite set (fixed_points_infinite), and every partial recursive function has infinitely many codes (padding, the Padding Lemma). So the fixed points of self-application are many, and so are the codes of any one behaviour.

**Theorem: selfprints_behaviour_injective (Kleene.lean § VIII)**

For self-printing codes $c_1$ and $c_2$: if eval $c_1$ = eval $c_2$ then $c_1 = c_2$.

Channel 0 reads each code's Gödel number back out of its behaviour, and the encoding is one-to-one.

Reading: computational self-reference has a uniqueness, at the level of behaviour. For self-printing codes this replaces the statement that computational quines are not unique; for the periodicity predicate IsComputationalQuine that statement stays true, since constant codes computing different functions all meet it. Both hold: infinitely many self-printing codes exist, and no behaviour belongs to two of them. Contrast padding, where infinitely many codes share one behaviour.

Lean: ZeroParadox.selfprints_behaviour_injective. Purity: propext, Classical.choice, Quot.sound (#print axioms in the PurityCheck section of ZeroParadox/Computability/Kleene.lean); the predicate SelfPrints itself carries the same three.

---

## Section II: KleeneStructure — Bridging AFA and Computation

### I. The Bridge

ZP-J's AFAStructure typeclass encodes AFA self-containment in type theory: a predicate selfMem, AFA uniqueness (quine_unique), and the bridge field bot_self_mem (⊥ is self-containing). T-EXEC follows: the Quine atom equals ⊥.

ZP-K's KleeneStructure extends AFAStructure with a computational witness: a code botCode satisfying the selfApply periodicity condition, whose existence is guaranteed by Kleene's theorem. The class was built for the reading that AFA self-containment (the Quine atom, CC-2) and the Kleene fixed point are one structural role in two formal languages. That role is stated once, as the Quine-atom role Motion.SelfOnlyAt (a point that steps to itself and to nothing else), and each chart carries its own occupancy theorem: with membership as the step, ⊥ is the only occupant in an AFAStructure lattice (quineAtomHost_quine_role_iff, host afaStructure_isQuineAtomHost); on codes under the eval step no single code occupies it (evalStepMotion_not_selfOnlyAt), and for a computable map respecting equal output a class of codes up to equal output does (codes_vs_classes). No equation across a code and ⊥ is stated, and no field of the class asserts one.

| KleeneStructure Typeclass (Kleene.lean § II) |
| --- |
| class KleeneStructure (L : Type*) [ZPSemilattice L] extends AFAStructure L with: |
| (inherited) selfMem : L → Prop — self-membership predicate |
| (inherited) quine_unique — AFA uniqueness |
| (inherited) bot_self_mem — ⊥ is self-containing |
| (new) botCode : Code — a code supplied at instantiation; not a distinguished code |
| (new) botCode_is_quine : IsComputationalQuine botCode — a CLASS FIELD, not a theorem: the periodicity condition eval c n = eval c (encode c + n), which constant codes also satisfy |
| (new) bot_self_mem_from_kleene : selfMem ⊥ — a RESTATEMENT of the inherited AFAStructure field bot_self_mem, adding no content beyond it. Not an implication from the Kleene side. |
| |
| Any KleeneStructure instance must supply both the AFA witness (bot_self_mem) and the computational witness (botCode with botCode_is_quine). Both are class FIELDS — assumptions discharged at instantiation, not results proved here. They are required together because the class was built for the reading that they are one structural fact; that reading is not derived within it, and its identity half is retired in favour of per-chart occupancy of the Quine-atom role (I above). |

### II. Why Not a Bridge Axiom?

No identification between AFA self-containment and Kleene computational fixed points is asserted, as a new axiom or as a class field. KleeneStructure is a typeclass: any concrete type must discharge its fields (among them a predicate selfMem with ⊥ its unique witness, and a code botCode meeting the periodicity condition) as proof obligations. They are checked at instantiation, not accepted globally.

> The distinction from ZP-J carries over: a freestanding axiom says "trust me." A typeclass field says "prove it for your specific type, or it does not compile." The MachinePhase instance in § V shows how the obligation is discharged concretely.

---

## Section III: T-COMP — The Three-Way Equivalence, and the Bundled Computational Condition

ZP-J T-EXEC established a three-way equivalence in a lattice carrying AFAStructure: Quine atom (in the lattice's AFA structure; in Mathlib's ZFSet the role is unoccupied, zfset_quine_role_unoccupied) ↔ bottom element (order-theoretic) ↔ join identity (algebraic). T-COMP, whose statement is (1) ↔ ((2) ∧ (3)), proves those three equivalent together with da2_bottom_characterization, which gives (2) ↔ (3). ZP-K bundles a fourth face — Kleene fixed point (computational) — as a KleeneStructure class field, so all four are present simultaneously in any KleeneStructure lattice, but the fourth is required at instantiation rather than proved equivalent to the others.

**Remark R-K.0 — What T-COMP Proves, and What "Four-Way" Meant**

The equivalence among (1)–(4) has two distinct sources:

(1)–(3) are equivalent by T-EXEC (ZP-J), with da2_bottom_characterization for (2) ↔ (3): any element that is a Quine atom is also ⊥ and a join identity, and vice versa. This is a genuine logical derivation — the three properties are proved to coincide from the AFAStructure axioms.

(4) is present in any KleeneStructure instance because KleeneStructure requires it as a typeclass field: botCode_is_quine must be supplied at instantiation. There is no independent proof that satisfying condition (1) (being a Quine atom in the AFA sense) entails satisfying condition (4) (being a Kleene fixed point), or vice versa. The two are combined by the typeclass definition — they are required together because we take them to be the same structural fact, not because one is derived from the other.

Stated at the level of types, which is where the gap is visible: conditions (1)-(3) are properties of one element q of the lattice L, and T-COMP relates them to each other. Condition (4) is a property of a Code. There is no function, and no equivalence, between Code and L anywhere in the development — so the fourth condition is not connected to the other three by a mapping; it is required alongside them by the class. da1_paths_unified records exactly this and no more: it is a conjunction of a fact about bot in L and a fact about botCode in Code.

In short: "four-way equivalence" means "all four hold in any KleeneStructure instance." (1)–(3) are independently proved equivalent. (4) is bundled in by the typeclass requirement. The philosophical claim — that Kleene computational self-reference and AFA set-theoretic self-reference are the same thing — is the motivation for the typeclass design, not a consequence derived within it. Its identity half is retired: what the two share is the Quine-atom role, whose occupancy is settled chart by chart (Section II.I), and that ⊥ actually executes is the occurrence commitment.

**Theorem T-COMP — Computational Grounding (Kleene.lean § III)**

In any KleeneStructure lattice L, for any q : L, the following are equivalent:

(1) IsQuineAtom q — the Quine-atom role in the lattice's AFA structure (CC-2)

(2) q = ⊥ — order-theoretic minimum (ZP-A)

(3) ∀ x : L, join q x = x — algebraic generator (ZP-A A4)

Separately, the KleeneStructure requirement: ∃ botCode : Code, IsComputationalQuine botCode (computational self-reference). It is not a clause of t_comp; it is present in any KleeneStructure instance because botCode_is_quine is a required field. The equivalence of (1)–(3) is derived by T-EXEC, with da2_bottom_characterization for (2) ↔ (3).

Lean: ZeroParadox.t_comp. Purity: standard foundational axioms — from Mathlib computability. ✓

## I. Why Four Languages?

DA-1's three informal paths (Path 1: AFA structural, Path 2: informational, Path 3: Kolmogorov/computational) were previously understood as three separate corroborations converging on the same conclusion. In ZP-E each argues for DA-1's precondition, which is what the occurrence commitment asserts, and none derives it. Path 1's Lean counterpart is da1_closed_concrete, which proves IsQuineAtom

(⊥ : MachinePhase) and nothing computational; Path 3's Lean counterpart is the machinePhaseKleene instance's botCode_is_quine field, a KleeneStructure requirement that constant codes also meet, not a witness of execution and not a second independent proof.

Path 1 says: nothing external to ⊥ can execute ⊥, so if ⊥ executes at all, the executor is ⊥ itself, ⊥ read as the Quine atom (CC-2); that rules out an external executor, not an inert ⊥. Path 3 says: no shorter external program generates ⊥; that rules out a shorter external generator, not an inert string, so Path 3 does not derive executing from incompressibility. The framework READS these as one claim in two vocabularies. What is proved is weaker and is a conjunction, not an identity: both hold together in any KleeneStructure. They cannot be equated formally — one is a statement about an element of the lattice, the other about a Code, and an equation across those types is not a well-formed proposition.

**Theorem: da1_paths_unified (Kleene.lean § IV)**

In any KleeneStructure lattice:

IsQuineAtom ⊥ ∧ IsComputationalQuine botCode

A CONJUNCTION: the AFA self-containment argument (Path 1) and the Kleene computational fixed-point argument (Path 3) both hold, simultaneously witnessed by the KleeneStructure instance. That they are the SAME fact is a framework reading, not a clause of this theorem — the second conjunct is the class field botCode_is_quine.

Lean: ZeroParadox.da1_paths_unified. Purity: standard foundational axioms only. ✓

## II. Description versus Execution

This section asks whether mathematical self-reference implies computational execution: whether a self-referential ⊥ is running (Sense B, ZP-E DA-1 insert § II) rather than an inert description (Sense A).

On the framework's reading, ⊥ in the computational instantiation is read as the universal Turing machine in its ground state, which is not a description awaiting an external executor: if it executes at all, it executes itself (Path 1, the framework's requirement, not a theorem). That rules out an external executor, not an inert ⊥: on MachinePhase, ⊥ is the Quine atom (da1_closed_concrete), and T-SNAP holds in a dynamics in which nothing leaves $c_0$ (tsnap_holds_but_nothing_moves). That it does execute, so that the configuration reaching $P_0$ is running, is the occurrence commitment, which DA-1 consumes and does not supply (ZP-E).

**Theorem: description_instantiation_gap_closed (Kleene.lean § IV)**

In any KleeneStructure lattice:

IsQuineAtom ⊥ ∧ ∀ q : L, IsQuineAtom q → q = ⊥

On the framework's reading, ⊥ is not a description awaiting an external interpreter: if it executes, it is its own executor, the universal machine in its ground state. Lean proves only the Quine-atom statement above; that ⊥ executes is the occurrence commitment, not this theorem. Lean witnesses ⊥ as the AFA Quine atom of MachinePhase (da1_closed_concrete) and carries the Kleene quine as a KleeneStructure requirement (botCode_is_quine); no Lean statement makes them one object, and what they share is the Quine-atom role, whose occupancy is settled chart by chart (Section II.I).

Lean: ZeroParadox.description_instantiation_gap_closed. Purity: standard foundational axioms only. ✓

A description has two uses. Cotler, Hongler and Hudcová (arXiv:2510.08342, 2025, p. 2) report von Neumann's self-replicating cellular automaton (their ref. [1]) as "leveraging the dual use of an organism's description – once to be interpreted for construction and once to be transcribed to the new offspring". Reading: the transcribed use is Sense A, a description copied as data, and the interpreted use is Sense B, a description being run (ZP-E, DA-1 insert § II). A description present on a tape supplies the first use and not the second.

Local universality does not give replication. The same authors construct a "non-talking heads" cellular automaton that is locally Turing-universal and yet cannot sustain non-trivial self-replication (their Theorem 2.8, p. 8): "any head encountering a cell marked by another head halts immediately. This blocking of information transfer prevents self-replication" (p. 7). Their Theorem 2.9 (p. 8) places self-replication strictly between two strengths of universality, GloballyUniversal ⊊ UniversalSelfReplicating ⊊ LocallyUniversal, and they note that "a quine produces a description of the copying mechanism without replicating the mechanism itself" (p. 8). Reading: the self-printing universal codes of Section I.III are quines in that sense. Each returns a description of itself, its own Gödel number, on channel 0: the description in Sense A.

In one example, one observable does not show occurrence. carry_steps_onward_forever_yet_shows_nothing (Occurrence.lean § VI-c): a deterministic step that, at every state, moves to a different state, while its observable, the first component of the state, stays constant on everything reachable. Reading: an observer of that observable sees the same constant trace as from a dynamics in which nothing moves (stuckPhase, tsnap_holds_but_nothing_moves), so for that example watching that observable does not settle whether it is running. Lean states this for carry and its one observable, not for every observable.

Reading, in two charts, neither denied. History chart: a complete description of a run, every configuration in order, IS the run; nothing further is left to happen. State chart: a description, even a complete one, is a configuration, and that it is running is a further fact, the occurrence commitment, which DA-1 consumes and does not supply (ZP-E). Both are readings.

---

## Section IV: Axiom Footprint

This section records measurements, not a rule. `#print axioms` traverses a declaration’s statement as well as its proof term and reports every axiom it reaches from either. Where the statement reaches an axiom, every proof of that statement carries it; where only the proof does, whether another proof avoids it is a separate question, and the table does not answer it. The following were measured on 2026-09-19 against the pinned Mathlib; re-run them rather than citing this table if the answer matters. On this toolchain `#print axioms` can under-report for declarations imported from another module (lean4 issue #15226); for the "none" rows, the issue’s own signature was checked and did not fire.

| Declaration | Axiom footprint |
|---|---|
| Nat.Partrec.Code | none |
| Nat.Partrec.Code.eval | none |
| IsKleeneFixedPoint | none |

| Declaration | Axiom footprint |
|---|---|
| encodeCode_self(encodeCode c = encodeCode c, by rfl) | none |
| encode_self(Encodable.encode c = Encodable.encode c, by rfl) | propext, Classical.choice, Quot.sound |
| kleene_fixed_point_exists | propext, Classical.choice, Quot.sound |
| roger_fixed_point_exists | propext, Classical.choice, Quot.sound |
| machinePhaseKleene | propext, Classical.choice, Quot.sound |
| da1_closed_concrete | propext, Classical.choice, Quot.sound |
| machinePhaseAFA | none |
| bot_is_quine_atom | none |
| t_exec | none |

Two readings of that table are worth naming, and neither generalises beyond the rows above. The two spellings of a code's Gödel number differ: the raw encodeCode is axiom-free, while Encodable.encode reaches through Mathlib's Denumerable Code instance, whose own term spends Classical.choice. And one statement is proved both ways in Kleene.lean §V — IsQuineAtom (bot : MachinePhase) via machinePhaseKleene carries the triple, and the same statement via machinePhaseAFA carries nothing. The axioms do not enter through ZPSemilattice or AFAStructure.

The Kleene.lean § VIII and Disjunctive.lean declarations cited in Section I.III and Section VI are not in that table. Their #print axioms lines are in those two files' PurityCheck sections; those of kleene_cf and selfref_universal_exists_enc are in the PurityCheck sections of ZeroParadox/Computability/KleeneChoiceFree.lean and ZeroParadox/Computability/UniversalChoiceFree.lean. Where Classical.choice enters selfref_universal_exists is recorded in ZeroParadox/Computability/Kleene.md § VIII.

ZP-J T-EXEC and all its corollaries remain axiom-free. Classical.choice is not confined to the computability rows: in Disjunctive.lean, champ_disjunctive carries it, reached through disjunctive_iff_once, while the at-least-once form champ_disjunctiveOnce measures propext and Quot.sound only (#print axioms in the PurityCheck section of ZeroParadox/Information/Disjunctive.lean).

**Remark: Classical Choice in Computability**

Kleene's theorem as Mathlib states it, $\forall$ f, Partrec$_2$ f $\to$ $\exists$ c, eval c = f c, carries Classical.choice in its statement: that statement, measured as a proposition, reports propext, Classical.choice and Quot.sound, and so does the statement of Rogers' theorem, $\forall$ f, Computable f $\to$ $\exists$ c, eval (f c) = eval c. So every proof of either, as stated, carries the triple. Where it enters each statement is recorded in ZeroParadox/Computability/Kleene.md § VIII. A restatement of Kleene's form whose statement and proof are both choice-free, for a transformation given as a code rather than as a Partrec$_2$ function, is kleene_cf (ZeroParadox/Computability/KleeneChoiceFree.lean, propext and Quot.sound only; the Open Items Register below); Rogers' form has no such restatement in this corpus.

**Remark: Classical Choice in Computability**

The MachinePhase instance (§ V) picks botCode with Classical.choose from computational_quine_exists, which is what makes machinePhaseKleene noncomputable. A computable instance with the constant code Code.const 0 also exists (the example after machinePhaseKleene in Kleene.lean § V).

---

# Section V: MachinePhase Instance — DA-1: what Lean witnesses

## I. The Concrete Instantiation

ZP-E's MachinePhase type is the two-element type {initial, running} carrying the ZPSemilattice instance (bot = initial = $c_0$, join = binary maximum). ZP-J gave it AFAStructure via the selfMem predicate. ZP-K gives it KleeneStructure by adding the computational witness botCode.

The selfMem definition for MachinePhase is: selfMem x := x = ⊥. This is the CIC-compatible encoding of AFA self-containment: "self-containing" means "equals the bottom element." Anti-foundation is not required at the typeclass level — the relevant structural fact (⊥ is the unique self-containing element) is captured by the definition and proved by rfl.

| AFAStructure MachinePhase Instance (Kleene.lean § V) |
|---|
| instance machinePhaseAFA : AFAStructure MachinePhase where |
| selfMem x := x = ⊥ (self-containing = equals initial state) |
| quine_unique _ _ hx hy := hx.trans hy.symm (if x = ⊥ and y = ⊥ then x = y) |
| bot_self_mem := rfl (⊥ = ⊥, proved by reflexivity) |
| |
| This is the CIC-compatible stand-in for the Quine atom (CC-2): the initial machine state is self-containing and is the unique element with this property, so with membership as the step it is the only occupant of the Quine-atom role (quineAtomHost_quine_role_iff, host afaStructure_isQuineAtomHost). No ZF+AFA axiom is added — the structural fact is encoded as a definition that Lean verifies. |

| KleeneStructure MachinePhase Instance (Kleene.lean § V) |
|---|
| noncomputable instance machinePhaseKleene : KleeneStructure MachinePhase where |
| botCode := Classical.choose computational_quine_exists |
| botCode_is_quine := Classical.choose_spec computational_quine_exists |
| bot_self_mem_from_kleene := rfl |
| |

| KleeneStructure MachinePhase Instance (Kleene.lean § V) |
| --- |
| botCode is a class field supplied at instantiation; this instance supplies it as Classical.choose applied to computational_quine_exists — some witness of that existence statement, not a distinguished code. Reading it as "the program that IS its own program" is not a property this instance establishes, and no field of the class asserts it (its identity half is retired, Section II.I): IsComputationalQuine is a periodicity condition, strictly weaker than self-reference. |

## II. DA-1: what Lean witnesses

With the MachinePhase KleeneStructure instance in place, the abstract theorem da1_computational (which holds for any KleeneStructure lattice) applies directly to ZP-E's machine. The result is concrete.

| Theorem da1_closed_concrete — Path 1's Lean counterpart (Kleene.lean § V) |
| --- |
| da1_closed_concrete : IsQuineAtom (⊥ : MachinePhase) |
| The initial machine state $c_0$ is a Quine atom: it is self-containing and is the unique self-containing element of the MachinePhase lattice. |
| Interpretation (the framework's reading, not a Lean theorem): $c_0$ is read as the universal Turing machine in its ground state, needing no external executor: if it executes at all, it executes itself (Path 1). That it executes is the occurrence commitment, which DA-1 consumes, not a consequence of this theorem. |
| Lean: ZeroParadox.da1_closed_concrete. Purity: standard foundational axioms only (the same statement has an axiom-free proof, bot_is_quine_atom). ✓ |

## III. What Changed for DA-1

ZP-E's DA-1 section previously carried the designation "Outside Lean Scope" with three justifications: Path 1 requires ZF+AFA, which ZP-E held incompatible with Lean's CIC/MLTT; Path 3 requires Kolmogorov complexity (it is uncomputable, and that alone does not keep it out of Lean, where a noncomputable definition is allowed; no formalization of it was located in the pinned Mathlib as of 2026-10-04, searched as recorded in ZeroParadox/Information/Disjunctive.md, and Lean developments outside Mathlib that define it are named at the ZPSurprisal entry of LEAN_CUSTOM_REGISTRY.md); Path 2 requires a step from unbounded surprisal to executing, which ZP-E names a bridge principle of its own, a missing principle and not a missing proof.

For Path 1 (AFA structural), in Lean the two charts split: Mathlib's ZFSet, a model of ZFC, contains no set that is a member of itself (no_quine_atom; Mathlib's ZFSet.mem_irrefl), while outside ZFSet Lean hosts a one-atom model whose distinguished element is its own unique self-member (the QuineHost instance on OneAtom, ZeroParadox/Settheory/QuineHost.lean § V). The Lean counterpart of Path 1 is da1_closed_concrete, which proves IsQuineAtom (⊥ : MachinePhase) and nothing computational: in the AFAStructure typeclass, selfMem stands in for the Quine atom (CC-2) in CIC-compatible form, and the proof obligation is discharged by the MachinePhase instance. Path 3 (computational) is NOT resolved here. Its Lean counterpart, botCode_is_quine, is a KleeneStructure class field — an assumption supplied at

instantiation, not a second independent proof — and the condition it requires, IsComputationalQuine, is a periodicity condition that a constant code satisfies trivially. The Kolmogorov reading ("no shorter program is prior to ⊥") has no formal content in ZP-K: Kolmogorov complexity is not formalized in this corpus (a scope decision, recorded at the ZPSurprisal entry of LEAN_CUSTOM_REGISTRY.md). Path 3 therefore derives nothing here: in ZP-E it bears on DA-1's precondition and does not derive executing from incompressibility, and that precondition is what the occurrence commitment asserts.

Path 2 (informational: unbounded surprisal → executing) is a bridge principle of its own — a missing principle, not a missing proof — which the framework does not adopt, which is not the occurrence commitment, and which is not a premise of the Snap. The mathematics of L-INF (l_inf) is proved; but the step from "exceeds every finite informational bound" to "therefore executing" asks what it means for a mathematical structure to instantiate rather than merely satisfy conditions. No computability library answers this question. Importantly, DA-1 does not depend on Path 2: the formal spine (DP-2 + da1_minimal_path) is proved axiom-free. Path 2 is motivational context.

DA-1 Lean scope status after ZP-K: Path 1 (structural, AFA): Lean counterpart da1_closed_concrete : IsQuineAtom (⊥ : MachinePhase), nothing computational; it rules out an external executor, not an inert ⊥. Path 3 (computational, Kleene): botCode_is_quine is a KleeneStructure class field, assumed at instantiation; Path 3 does not derive executing from incompressibility. Path 2 (informational, L-INF): UNADOPTED BRIDGE PRINCIPLE — a missing principle, not a missing proof, and not the occurrence commitment. None derives DA-1's precondition, which is what the occurrence commitment asserts.

---

## Section VI: Replication — Presence, Address, and Selection

This section follows one arc: every instruction is present on a disjunctive tape (I), self-printing interpreters differ only by an address (II), a disjunctive tape holds no full copy of itself (III), and constructing an address is a computation while selecting a code by its behaviour is where choice would work (IV, a reading). V adds the zero tape and VI the replication setting of Cotler, Hongler and Hudcová. Presence on a tape is Sense A (ZP-E, DA-1 insert § II); that anything runs is the occurrence commitment, as in Section III.II.

### I. Presence: Every Instruction Is on the Tape

A tape is disjunctive (the standard term) when every finite word occurs in it. Barnsley and Leśniak (arXiv:1203.0481v2, § 3, p. 6) define it that way and note that every finite word then occurs infinitely often (their remark and Prop. 1, same page); Lean defines the infinitely-often form (Disjunctive) and proves the two forms agree (disjunctive_iff_once). On a disjunctive tape the binary of every program code occurs infinitely often (code_occurs_of_disjunctive). That is presence of the description, Sense A; nothing in the statement reads or runs a code. Disjunctive does not need randomness: champ, a Champernowne-style tape, is disjunctive and primitive recursive (champ_disjunctive, champ_primrec), and almost every fair-coin tape is disjunctive (fairTape_disjunctive_ae).

The commitment, stated on its own: the framework holds that the occupant of its ⊥ role, ⊥ of a ZPSemilattice, read as a tape, is maximally complex. That is a commitment about that occupant, not about any one tape named here, and in particular not about the all-false tape, ⊥ of the pointwise tape order (V below), which carries no code word but Code.zero's empty one. No map from a ZPSemilattice to ℕ → Bool is claimed (ZeroParadox/Information/Disjunctive.lean § VI).

The standard theory, stated separately and not proved in Lean: with K the prefix-free Kolmogorov complexity, a sequence x has K(first n bits of x) ≥ n − c for some constant c and every n exactly when it is Martin-Löf random (the Levin–Schnorr theorem; Franklin and Porter, arXiv:2004.02851, Theorem 2.5, crediting Levin and Schnorr), and Martin-Löf random sequences are disjunctive. The prefix-free form is needed: with plain complexity C, every infinite sequence A has C(first n bits of A) ≤ n − log n for infinitely many n (Franklin and Porter, arXiv:2004.02851, § 2.2, p. 15, crediting Martin-Löf). Martin-Löf (Z. Wahrsch. verw. Geb. 19 (1971) 225–230, Theorem 1) proves the form conditional on the length n, for plain conditional complexity: for every sequence x and every recursive f whose series of terms 2^−f(n) diverges, C(first n bits of x | n) < n − f(n) for infinitely many n. Martin-Löf randomness and Kolmogorov complexity are not located in the Mathlib pin as of 2026-10-04 (searches recorded in ZeroParadox/Information/Disjunctive.md). Disjunctive does not imply random: champ is disjunctive and computable.

Reading: if the occupant of the framework's ⊥ role is read as a tape ℕ → Bool, and its maximal complexity is read as the Levin–Schnorr criterion above, every prefix incompressible up to a constant, K(first n bits of x) ≥ n − c, then the standard theory would make that tape Martin-Löf random and so disjunctive, and every instruction would be present on it.

## II. Two Self-Printing Interpreters Differ Only on Channel 0

**Theorem: selfPrints_universal_address (Kleene.lean § VIII)**

For self-printing universal codes c and d: they agree on every interpreter channel e + 1, and if c ≠ d they differ on channel 0, where each returns its own Gödel number.

The control beside it in Kleene.lean shows that Universal carries the first half: Code.zero is self-printing and not universal, and it disagrees with a self-printing universal code on a channel e + 1.

Reading: replicas share everything but the address. Incompressibility is a property of each instance, and the next instance adds only its address. That is a claim about relative complexity, and it stays a reading while Kolmogorov complexity is not formalized in this corpus (a scope decision, recorded at the ZPSurprisal entry of LEAN_CUSTOM_REGISTRY.md, which also names Lean developments outside Mathlib that define it).

Lean: ZeroParadox.selfPrints_universal_address. Purity: propext, Classical.choice, Quot.sound (#print axioms in the PurityCheck section of ZeroParadox/Computability/Kleene.lean).

## III. No Full Self-Copy on a Disjunctive Tape

A disjunctive tape equals its own shift by no offset a > 0 (disjunctive_not_periodic; not_disjunctive_of_periodic is the same fact read from the periodic side). At offset 0 every tape equals its own shift, champ included (the control example beside it in Disjunctive.lean § V). Barnsley and Leśniak (arXiv:1203.0481v2, pp. 7-8) note that "a disjunctive sequence cannot be almost periodic", almost periodic in their sense meaning that each word occurring infinitely often occurs in every segment of some length m, with m depending on the word. A tape equal to its own shift by some a > 0 is almost periodic in that sense (the example after not_disjunctive_of_periodic in Disjunctive.lean § V), so their remark is the stronger statement and disjunctive_not_periodic is its periodic special case. Reading: a tape carrying every code word holds no full copy of itself at any positive offset; copies sit side by side, told apart by an address (II).

## IV. Constructing an Address, and Selecting One

Constructing an address is the recursion theorem's fixed point of a partially computable transformation: selfref_universal_exists takes its code from Mathlib's fixed_point$_2$ applied to a partially computable map (selfPrintOrDelegate_partrec). The selection results in the corpus are about two-element charts, Chart = Bool (ZeroParadox/Valuation/PoleChartSelection.lean): an inhabited predicate on Bool that carries a DecidablePred instance has a witness computed by if (select_of_decidable), and a uniform selector for every inhabited predicate on Bool is, definitionally, the choice fragment (uniformChartSelection_iff_choiceFragment). A DecidablePred instance is not an algorithm: Classical.decPred supplies one for every predicate, through choice. Neither result is about selecting a code.

Reading: constructing an address is a computation, and selecting a code by its behaviour is where choice would work, by analogy with the two-element results above, which are stated for Bool and not for codes. The Lean footprint does not draw that line, and is stated as measured: selfref_universal_exists carries Classical.choice, and so does its statement, through Mathlib's numbering of codes (Denumerable Code, reached through ofNatCode); restated with encodeCode for the Gödel number and the channel of code d written Nat.pair (encodeCode d + 1) n, the statement is axiom-free, and it has a proof at propext and Quot.sound only, selfref_universal_exists_enc (ZeroParadox/Computability/UniversalChoiceFree.lean). ZeroParadox/Computability/Kleene.md § VIII records the measurements. Every proof of the statement as written carries choice.

## V. The Zero Tape: Reference and Self

In $\mathbb{N} \to$ Bool under the pointwise order, $\bot$ is the all-false tape, and that tape is not disjunctive (allFalse_not_disjunctive); it carries the word of no code other than Code.zero, whose word is empty (allFalse_misses_code, codeWord_zero_width). Under pointwise exclusive-or, every tape's self-difference is the all-false tape, and read against the all-false tape a tape is itself, pointwise by Bool.xor_false (the examples in BottomCannotBe.lean, "The infinitude in two charts: content and self", apply it to champ).

Reading, in two charts, neither denied: the all-false tape, $\bot$ of the pointwise tape order, is the reference every comparison runs through (content chart) and every tape's self-difference (self chart), one object in two charts. That the occupant of the framework's $\bot$ role, $\bot$ of a ZPSemilattice, read as a tape, is maximally complex is a commitment, stated as no equation with any one tape: it is not a statement about this all-false tape, which is not disjunctive and carries no code word but Code.zero's empty one.

## VI. Replication Beyond Self-Reference, in Cotler, Hongler and Hudcová's Setting

In Cotler, Hongler and Hudcová's setting, information crossing between heads is, in their word, crucial: "inter-head communication is crucial for replication" (p. 8). Their non-talking heads automaton blocks that crossing and is locally universal without self-replication (Theorem 2.8, p. 8). A second capacity separates the two strengths of universality, GloballyUniversal $\subsetneq$ LocallyUniversal (their Theorem 2.3, p. 5): of the reversible automata, "Some can locally implement reversible universal Turing machines but cannot globally simulate irreversible CAs". Reading: these are two different capacities, inter-head communication, which the authors call "crucial" for replication and which the non-talking-heads rule blocks by halting any head that encounters a cell another head has marked, and irreversible erasure, which such a reversible automaton cannot simulate globally. This mapping onto the framework is a reading.

---

## Traceability Register — ZP-K

| Claim | Grounded In | Axioms | Status |
|---|---|---|---|
| selfApply_partrec | eval_part (Mathlib) + Primrec.encode + Primrec.nat_add | Standard Mathlib foundational axioms | Lean: selfApply_partrec ✓ |
| computational_quine_exists | kleene_fixed_point_exists + selfApply_partrec | Standard Mathlib foundational axioms | Lean: computational_quine_exists ✓ |
| T-COMP: three-way equivalence | ZP-J T-EXEC (t_exec_triple_iff) | Standard Mathlib foundational axioms | Lean: t_comp ✓ |
| da1_paths_unified | bot_is_quine_atom + botCode_is_quine | Standard Mathlib foundational axioms | Lean: da1_paths_unified ✓ |
| description_instantiation_gap_closed | bot_is_quine_atom + ZP-J t_exec | Standard Mathlib foundational axioms | Lean: description_instantiation_gap_closed ✓ |
| machinePhaseAFA (AFAStructure) | selfMem := x = ⊥; quine_unique; bot_self_mem := rfl | No axioms | CIC stand-in for the Quine atom (CC-2) on MachinePhase ✓ |
| machinePhaseKleene (KleeneStructure) | machinePhaseAFA + Classical.choose computational_quine_exists | Standard Mathlib foundational axioms | noncomputable — classical choice for botCode ✓ |
| da1_closed_concrete | da1_computational + machinePhaseKleene | Standard Mathlib foundational axioms | Lean: da1_closed_concrete ✓ DA-1 structural half — no Code, no execution (definitionally: under selfMem x := x = ⊥, reduces to (⊥ = ⊥) ∧ (∀ x, x = ⊥ ⇒ x = ⊥); structural closure by typeclass design — see R-K.0) |
| fixed_points_infinite, padding | fixed_point$_2$_unbounded (Kleene.lean § VIII) | propext, Classical.choice, Quot.sound (#print axioms, ZeroParadox/Computability/Kleene.lean) | Lean ✓ — infinitely many fixed points for every partially computable F; the Padding Lemma |

| Claim | Grounded In | Axioms | Status |
|---|---|---|---|
| selfref_universal_exists, selfref_universal_infinite | fixed_point$_2$ (Mathlib) + selfPrintOrDelegate_partrec; fixed_points_infinite | propext, Classical.choice, Quot.sound (#print axioms, ZeroParadox/Computability/Kleene.lean), carried by the statement; restated form proved without choice, selfref_universal_exists_enc (ZeroParadox/Computability/Kleene.md § VIII) | Lean ✓ — existence of self-printing universal codes; nothing says a code is run |
| selfprints_behaviour_injective | SelfPrints at channel 0 + Encodable.encode_injective | propext, Classical.choice, Quot.sound (#print axioms, ZeroParadox/Computability/Kleene.lean) | Lean ✓ — Reading: a uniqueness at the level of behaviour |
| selfPrints_universal_address | SelfPrints + Universal + Encodable.encode_injective | propext, Classical.choice, Quot.sound (#print axioms, ZeroParadox/Computability/Kleene.lean) | Lean ✓ — Reading: replicas share everything but the address |
| code_occurs_of_disjunctive | Disjunctive (Disjunctive.lean § I) at each code word | propext, Classical.choice, Quot.sound (#print axioms, ZeroParadox/Information/Disjunctive.lean) | Lean ✓ — presence, Sense A; no code is read or run |
| champ_disjunctive, champ_disjunctiveOnce, champ_primrec | tri / untri slot layout + bitAt_ofBits; disjunctive_iff_once; Mathlib Primrec | champ_disjunctive: propext, Classical.choice, Quot.sound; champ_disjunctiveOnce: propext, Quot.sound; champ_primrec: propext, Classical.choice, Quot.sound (#print axioms, ZeroParadox/Information/Disjunctive.lean) | Lean ✓ — disjunctive without randomness |
| fairTape_disjunctive_ae | second Borel–Cantelli (measure_limsup_eq_one) on disjoint aligned blocks | propext, Classical.choice, Quot.sound (#print axioms, ZeroParadox/Information/Disjunctive.lean) | Lean ✓ — almost every fair-coin tape is disjunctive |
| disjunctive_not_periodic | not_disjunctive_of_periodic (Disjunctive.lean § V) | propext, Classical.choice, Quot.sound (#print axioms, ZeroParadox/Information/Disjunctive.lean) | Lean ✓ — no full self-copy at any offset $a > 0$ |

| Claim | Grounded In | Axioms | Status |
|---|---|---|---|
| carry_steps_onward_forever_yet_shows_nothing | stutter_obs_const + carry_stutters (Occurrence.lean § VI-c) | propext, Classical.choice, Quot.sound (#print axioms, ZeroParadox/Computability/Occurrence.lean) | Lean ✓ — steps onward forever with a constant observable |
| Reading: zero tape (Section VI.V) | BottomCannotBe.lean, "The infinitude in two charts: content and self" | — (interpretation) | Reading, not a theorem: the all-false tape as reference and as self-difference |
| Reading: complete description, two charts (Section III.II) | occurrence commitment (ZP-E) | — (interpretation) | Reading, not a theorem: history chart and state chart, both stated |

---

## Open Items Register — ZP-K

| Item | Status | Description |
|---|---|---|
| DA-1 Path 1 (AFA structural) | Lean counterpart proved; derives no execution | IsQuineAtom (⊥ : MachinePhase) — the structural half, Path 1's Lean counterpart. The theorem mentions no Code and no execution. Path 1 says that if ⊥ executes, it executes itself (the framework's requirement, not a theorem): it rules out an external executor, not an inert ⊥. That ⊥ executes is the occurrence commitment, which DA-1 consumes; Path 1 does not close it. |
| DA-1 Path 3 (computational) | ARGUES FOR THE PRECONDITION — derives nothing | botCode_is_quine is a KleeneStructure class field — assumed at instantiation, not proved. IsComputationalQuine is a periodicity condition satisfied by constant codes, and no Kolmogorov content exists in ZP-K. In ZP-E, Path 3 does not derive executing from incompressibility; DA-1's precondition is what the occurrence commitment asserts. |
| DA-1 Path 2 (informational) | UNADOPTED BRIDGE PRINCIPLE | L-INF (l_inf) is proved. The bridge "unbounded surprisal → executing" is a bridge principle of its own, a missing principle, not a missing proof; the framework does not adopt it, and it is not the occurrence commitment or a premise of the Snap. The gap between 'system at $P_0$' and 'system is running' cannot be closed by any computability library. DA-1 does not depend on Path 2. |
| selfApply uniqueness | CLOSED — two predicates, two answers | Codes meeting IsComputationalQuine are not unique: the constant codes alone give infinitely many (infinite_quine_family). That the Quine atom is ⊥ of the lattice, and the only one, flows from ZP-J T-EXEC (the AFA side). For self-printing codes, a uniqueness at the level of behaviour is proved (selfprints_behaviour_injective). |
| Rogers' fixed-point theorem | CLOSED — roger_fixed_point_exists | For any computable f : Code → Code, ∃ c, eval (f c) = eval c. Lean: roger_fixed_point_exists — standard foundational axioms only. ✓ |

| Item | Status | Description |
|---|---|---|
| ZP-B MachinePhase instance | OPEN — future work | The 2-adic model from ZP-B ($Q_2$ structure) has not been given a KleeneStructure instance. This is a natural extension but not required for T-SNAP or DA-1. |
| Choice-free recursion theorem | CLOSED for Kleene's form (kleene_cf) and the channel restatement (selfref_universal_exists_enc); Rogers' form not restated | The statement of selfref_universal_exists carries Classical.choice through Mathlib's numbering of codes (Denumerable Code), so every proof of it as written carries choice. Restated with encodeCode for the Gödel number and the channel of code d written Nat.pair (encodeCode d + 1) n, it is axiom-free, and selfref_universal_exists_enc (ZeroParadox/Computability/UniversalChoiceFree.lean) proves it at propext and Quot.sound only. The recursion theorem itself, for a transformation given as a code, is kleene_cf (ZeroParadox/Computability/KleeneChoiceFree.lean), at the same footprint. Rogers' form, ∀ f, Computable f → ∃ c, eval (f c) = eval c, has no choice-free restatement in this corpus. The record is ZeroParadox/Computability/Kleene.md § VIII. |
| Martin-Löf randomness ⇒ disjunctive | OPEN — not formalized in this corpus | Standard theory: a Martin-Löf random sequence is disjunctive, and by the Levin–Schnorr theorem Martin-Löf randomness is incompressibility of every prefix, K(first n bits of x) ≥ n − c for some constant c and every n, with K the prefix-free complexity (Franklin and Porter, arXiv:2004.02851, Theorem 2.5); with plain complexity C, every infinite sequence A has C(first n bits of A) ≤ n − log n for infinitely many n (Franklin and Porter, § 2.2, p. 15, crediting Martin-Löf; Martin-Löf 1971, Theorem 1, proves the form conditional on the length n). Martin-Löf randomness and Kolmogorov complexity are not located in the Mathlib pin as of 2026-10-04 (search record in ZeroParadox/Information/Disjunctive.md). Outside Mathlib, one development named at the ZPSurprisal entry of LEAN_CUSTOM_REGISTRY.md defines Martin-Löf randomness and proves the Levin–Schnorr theorem; a proof there that a Martin-Löf random sequence is disjunctive was not located as of 2026-11-01 (the search is recorded at that entry). Until this bridge is formalised, the framework's commitment that the occupant of its ⊥ role, ⊥ of a ZPSemilattice, read as a tape, is maximally complex reaches Disjunctive (Section VI.I) only as a reading. |

---

End of ZP-K | Computational Grounding of Self-Reference | DA-1 structural half: da1_closed_concrete : IsQuineAtom (⊥ : MachinePhase) - no code, no execution | Four-way equivalence (R-K.0): Quine atom, ⊥ and join identity proved to coincide; the Kleene fixed point is required by the typeclass, not derived | Path 2: an unadopted bridge principle, not a missing proof | All Kleene.lean theorems verified. Axioms: standard Mathlib foundational axioms.