Computation Precedes Naming

Naming Returns Computation

Larsen James Close

2026

Abstract

One equation on binary strings — fold ∘ unfold = id — forces structural results at each level of determination. The dual composition selfApp = unfold ∘ fold is idempotent and classifies every fold/unfold geometry into one of three regimes: selfApp = id (section closes), bounded (section does not close), or unbounded. All three are inhabited. The equation classifies — it does not choose.

Three independent results prove nonclosure on the classical carrier: no structure-preserving map from the frictionless model to the classical carrier (six lines, omega); selfApp ≠ id; and irreducible factoring through the header length. A fourth — the super-polynomial counting gap — establishes that the naming obstruction is intrinsic to the carrier's information geometry. The obstructions are geometric — fold shortens, unfold lengthens — and encoding-invariant. selfApp acts on programs, inputs, and certificates alike — any carrier element.

On the classical carrier, programs, inputs, certificates, and languages are all binary strings, definitionally. The complexity classes P and NP are defined through this fold/unfold — the structural results constrain the operations they are built from. The nonclosure is prior to the class distinction. The proofs apply at each instance. P ≠ NP on the classical carrier.

The obstruction is specific to classical geometry. Non-classical models remain open. The framework measures — it does not prescribe.

Machine-checked in Lean 4 with Mathlib (v4.28.0) (Close 2026; Moura and Ullrich 2021; The mathlib Community 2020): 22 source files, approximately 4,300 lines, zero sorry, zero custom axioms.

Contents

1/4 The question

The P versus NP problem asks whether every language that can be efficiently verified can also be efficiently decided — whether certificate access can always be eliminated.

A complexity class is a pairing of computational model and language. P pairs a decider with a language: given an input, the decider accepts or rejects in polynomial time. No certificate. NP pairs a verifier with a language and certificates: given an input and a certificate, the verifier checks in polynomial time. The question asks whether every language admitting the second pairing also admits the first.

On the classical carrier, programs, inputs, certificates, and languages are all binary strings. Both pairings operate through the same fold and unfold. Every computation on this carrier — every execution of every program — passes through these two morphisms.

1/2 Nonclosure

```
structure WeakReflData (A L : C) where
  fwd     : Hom (ihom A L) L
  bwd     : Hom L (ihom A L)
  fwd_bwd : fwd ∘ bwd = id
```

Retract and two morphisms. One equation.

The dual composition selfApp = bwd ∘ fwd is an endomorphism — idempotent, not necessarily identity. WeakReflData is sufficient for the entire computational grammar: self-application, fixed points, witness search, the recursion theorem. This is beta — the reach of one equation.

WeakReflData gives computation (beta). It has one equation. The dual composition exists — you can always compose bwd and fwd — but WeakReflData says nothing about it. Half the structure.

ReflData extends with the second equation: bwd ∘ fwd = id. Both compositions identity. That is eta — faithful naming, abstraction and evaluation as genuine inverses, zero information loss in the full round trip. Eta is not derived from the retraction. It is not a consequence of computation. It is a second equation, independent of the first.

The gap between WeakReflData and ReflData — between beta and eta, between computation and faithful naming — is the second equation. WeakReflData does not have it. Not as a failure. As an absence. On a specific carrier, the dual composition can be measured. On the classical carrier, measuring it gives selfApp ≠ id — the axiom would be false. Three independent proofs.

The regime question is whether the section closes: selfApp = id (Group C), bounded but not identity (Group B), or unbounded (Separation). The regime is geometric, determined by the fold/unfold.

When the two objects are the same type — the single-carrier specialization — fwd becomes fold, bwd becomes unfold, and both are carrier → carrier. This is a determination: it collapses two objects to one. The Lean codebase works at this level (RetractionModel, GRM). The structural results hold here. But WeakReflData is prior to this collapse.

The classical carrier has WeakReflData: the retraction holds. The section does not close: selfApp ≠ id when headerLen > 0. The gap is exactly headerLen bits — the irreducible cost of self-reference on this carrier. Three independent proofs of nonclosure are given in Section 1.

Every computational operation on the classical carrier passes through fold and unfold. The round trip through unfold then fold is selfApp. P and NP are defined on this carrier (`rfl`). The nonclosure — selfApp ≠ id, no structural section, irreducible drift — constrains every computation on it. The proofs apply at each instance.

1. Ten Lines

```
theorem fold_unfold_nonclosure (hne : h.headerLen > 0)
    (φ : FoldUnfoldSection (transportRM (classicalRM h))
                           (classicalRM h)) :
    False := by
  set R := classicalRM h
  set TR := transportRM R
  set t := Morphism.identity R
  have h_fold_id : TR.fold t = t := TR.roundtrip t
```

```
have h1 : R.fold (φ.map t) = φ.map t := by
  have := φ.map_fold t; rw [h_fold_id] at this; exact this.symm
have h2 : R.unfold (φ.map t) = φ.map t := by
  have := φ.map_unfold t; exact this.symm
have h_eq : R.fold (φ.map t) = R.unfold (φ.map t) := by rw [h1, h2]
have h_len : List.length (R.fold (φ.map t)) ≤ List.length (φ.map t) := by
  simp [R, classicalRM, tmFold, List.length_drop]
have h_len2 : List.length (R.unfold (φ.map t)) ≥
    List.length (φ.map t) + h.headerLen := by
  simp [R, classicalRM, tmUnfold, List.length_append, h.headerLen_eq]; omega
rw [h_eq] at h_len; omega
```

This is the engine: fold shortens, unfold lengthens, `omega`. No structure-preserving map can return from the frictionless model to the classical carrier. The type signature is pre-semantic — `classicalRM`, not `classicalGRM`; no grade function appears. Machine-checked in Lean 4 with Mathlib (The mathlib Community 2020). Zero `sorry`. Zero custom axioms. Only the standard kernel axioms (`propext`, `Classical.choice`, `Quot.sound`) — the same axioms Mathlib uses.

Four unconditional structural results characterize the classical carrier:

1. **selfApp factors through headerLen** (`classicalGRM_factorsThrough`, `RegimeClassification.lean:21`): self-reference costs exactly headerLen bits.
2. **No structural section** (`fold_unfold_nonclosure`, `Nonclosure.lean:27`): no `FoldUnfoldSection` from the frictionless model to the classical carrier. The proof above.
3. **Super-polynomial counting gap** (`construction_super_poly`, `CountingEngine.lean:128`): the endomorphism count exceeds description capacity at every polynomial.
4. **selfApp ≠ id** (`presemantic_not_group_C`, `Cascade.lean:276`): the section does not close. Pre-semantic.

Each is unconditional. Results 1, 2, and 4 each independently prove the section does not close on the classical carrier. Result 3 establishes that the naming obstruction is intrinsic to the carrier's information geometry. The nonclosure is prior to the complexity classes. P and NP are both defined on this carrier — Language = Set carrier is definitional (`language_is_carrier_set`, `rfl`). They inherit the carrier's constraints. The proofs apply at each instance. Section 3 shows the argument in full.

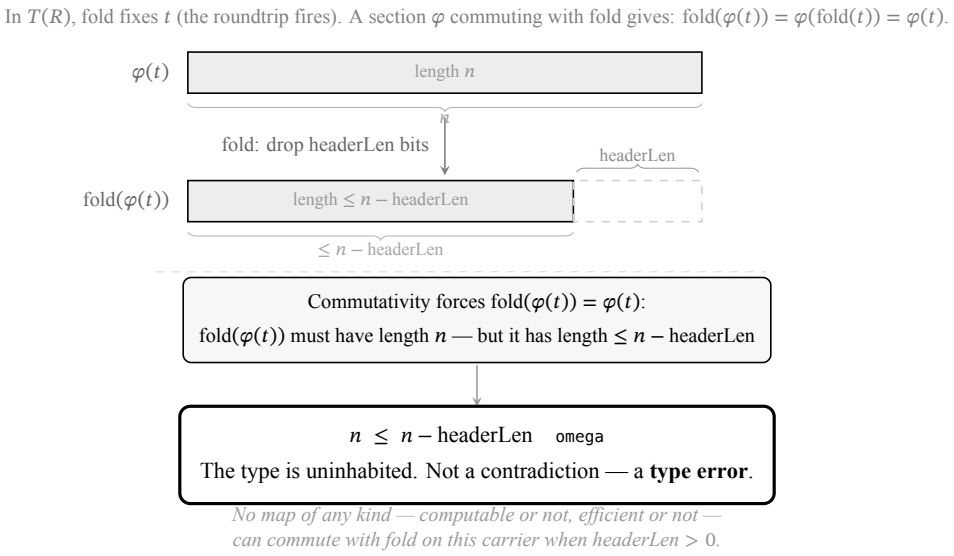


Figure 1: The type error. Fold drops `headerLen` bits (dashed portion). Commutativity forces $\text{fold}(\varphi(t)) = \varphi(t)$, but no string equals a shorter version of itself: $n \leq n - \text{headerLen}$ is impossible when headerLen > 0. The type is uninhabited by arithmetic.

One concrete consequence: selfApp maps both the rejection token [false] and the empty string [] to the acceptance token [true] (`selfApp_reject_displaced`, `selfApp_empty_displaced`, both `rfl`). The carrier's self-reference operation cannot distinguish acceptance from rejection. Arithmetic on binary strings, confirmed by Lean's kernel through definitional reduction.

The rest of this paper explains what every symbol means and why this arithmetic is about P vs NP. The compiled Lean project is 22 source files, approximately 4,300 lines. The one shown above (`fold_unfold_nonclosure`) is the simplest. It uses no grade function, no counting argument, no complexity-theoretic vocabulary. Pure carrier geometry.

2. What Every Symbol Means

2.1 The carrier

Binary strings:

```
abbrev BinString := List Bool
```

(`BinString.lean:22`)

Two operations on this carrier:

- **fold** = tmFold: drop the first `headerLen` bits. Shortens by `headerLen`. (`BinString.lean:62`)
- **unfold** = tmUnfold: prepend the header. Lengthens by `headerLen`. (`BinString.lean:66`)

`concreteHeader` = [true], so `headerLen` = 1. A `classicalGRM` bundles the carrier, fold, unfold, and grade = string length into a Graded Reflexive Model (`Instantiation.lean:30`).

2.2 The one equation

$$\text{fold}(\text{unfold}(x)) = x \quad \text{for all } x.$$

Prepend the header, then drop it. The string is restored. This is the retraction equation, and it always holds (`tmFold_tmUnfold`, `BinString.lean:71`).

The dual composition:

$$\text{selfApp}(x) = \text{unfold}(\text{fold}(x))$$

Drop the header, then prepend it. This replaces the first `headerLen` bits with the header. When `headerLen` > 0, this is NOT identity: selfApp([]) = [true] $\neq$ [].

Two morphisms. One composition is identity. The other — selfApp — is the structural endomorphism of the carrier.

2.3 The structural endomorphism

selfApp partitions the carrier. Every binary string is either fixed by selfApp (it already starts with the header) or displaced (selfApp replaces its first bits with the header). On the classical carrier with `headerLen` $= 1$: exactly half the strings at each positive grade are fixed, half are displaced (`selfApp_two_to_one`, `rfl`). The partition is total.

Programs, inputs, and certificates are carrier elements. They are all subject to selfApp. The structural results (Section 1) are unconditional properties of this endomorphism on this carrier. The nonclosure proofs apply at each instance. Section 3 shows the argument in full.

2.4 The proof, line by line

Let φ be a `FoldUnfoldSection` from $T(R)$ to R, where R is the classical `RetractionModel` (pre-semantic — no grade). $T(R)$ is the *transport model* — the version of R where selfApp $=$ id (the frictionless carrier). A `FoldUnfoldSection` commutes with fold and unfold. It is the weakest structural map — no grade condition, no computability requirement (`Transport.lean:68`).

1. Let t be the identity morphism in $T(R)$. In $T(R)$, fold fixes t (the roundtrip fires).
2. By φ's commutation with fold: $R.\text{fold}(\varphi(t)) = \varphi(t)$. Fold fixes $\varphi(t)$.
3. By φ's commutation with unfold: $R.\text{unfold}(\varphi(t)) = \varphi(t)$. Unfold fixes $\varphi(t)$.
4. Therefore fold and unfold agree on $\varphi(t)$: $R.\text{fold}(\varphi(t)) = R.\text{unfold}(\varphi(t))$.
5. But fold shortens by `headerLen` (`List.length_drop`). Unfold lengthens by `headerLen` (`List.length_append`).
6. When `headerLen` > 0: $|\varphi(t)| + \text{headerLen} \leq |\varphi(t)|$. Contradiction. `omega`.

$\varphi(t)$ would need to be simultaneously immune to shortening and lengthening. Nothing is.

2.5 The ruler

Take a ruler. *Fold* snaps off the first h inches from the left end. *Unfold* glues h inches back onto the left end. Glue-then-snap restores the ruler — that is the retraction equation, and it always works.

Now snap-then-glue: does it also restore the ruler? It does not — the inches that glue back are the header's inches, not the ones you snapped off. The ruler is changed. That is selfApp.

`fold_unfold_nonclosure` asks a sharper question. Imagine a *frictionless ruler* (the transport model) where snap-then-glue trivially works because $h = 0$ — nothing gets snapped, nothing gets glued. Could there be a structure-preserving way to map from the frictionless ruler to a real ruler ($h > 0$)? Any mapped ruler must survive both the real snap (which shortens by h) and the real glue (which lengthens by h) unchanged. When $h > 0$, no ruler can be simultaneously immune to shortening and lengthening. Six lines. `omega`.

2.6 The structural chain

The classical carrier has headerLen $= 1$. selfApp $\neq$ id (`presemantic_not_group_C`). No `FoldUnfoldSection` from $T(M)$ to M (`fold_unfold_nonclosure`). selfApp factors through headerLen (`classicalGRM_factorsThrough`). The endomorphism gap is super-polynomial (`construction_super_poly`). Four structural facts. Three independently prove nonclosure on the classical carrier. The counting gap establishes that the naming obstruction is intrinsic to the carrier's information geometry.

`selfApp_id_gives_section` (`Parallel.lean:175`) shows the positive direction: selfApp $=$ id would force headerLen $= 0$ and yield a section. The classical geometry does not admit this.

	Retraction	Section
Composition	fold $\circ$ unfold	unfold $\circ$ fold
Result	identity (axiom)	selfApp
Status	Always identity	Not automatically identity
Cost	Zero	headerLen
Group C	identity	identity
Classical (headerLen > 0)	identity	$\neq$ identity

What is genuinely open. P $=$ NP remains open for non-classical computation. `headerLen = 0` gives Group C (selfApp $=$ id, no obstructions). Group C is inhabited (`group_C_inhabited`, `Regime.lean:113`). The framework classifies all fold/unfold geometries. The classical carrier is one determination path. The framework measures — it does not prescribe.

3. Why This Arithmetic Is About P vs NP

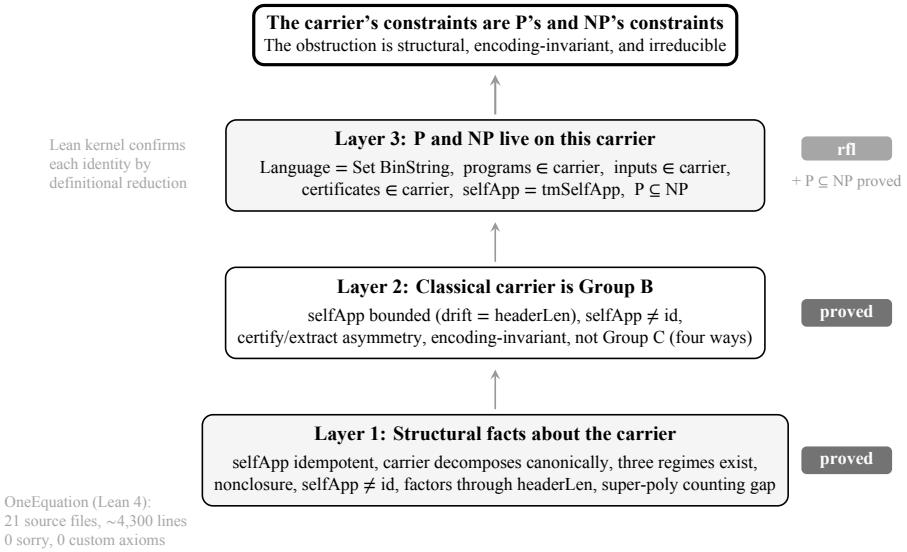


Figure 2: The claim ladder. Layer 1: structural facts about the carrier (machine-checked). Layer 2: regime classification (machine-checked). Layer 3: P and NP are definitionally on this carrier (rfl). The carrier's constraints apply to P and NP because they are the same objects on the same type.

3.1 One type

```
Language := Set BinString                                -- Language.lean:30
classicalGRM.carrier := BinString                        -- Instantiation.lean:30
language_is_carrier_set : Language = Set carrier := rfl -- CarrierConnection.lean:35
```

Languages are sets of carrier elements — definitionally, confirmed by the kernel (`rfl`).

```
classical_tm_on_carrier : classical_tm.run = concreteRun
    at carrier type := rfl                               -- CarrierConnection.lean:43
selfApp_eq_tmSelfApp : selfApp x = tmSelfApp x := rfl -- CarrierConnection.lean:53
```

Programs are carrier elements. Inputs are carrier elements. Certificates are carrier elements. The computation model operates on carrier elements. selfApp acts on all of them. There is one type.

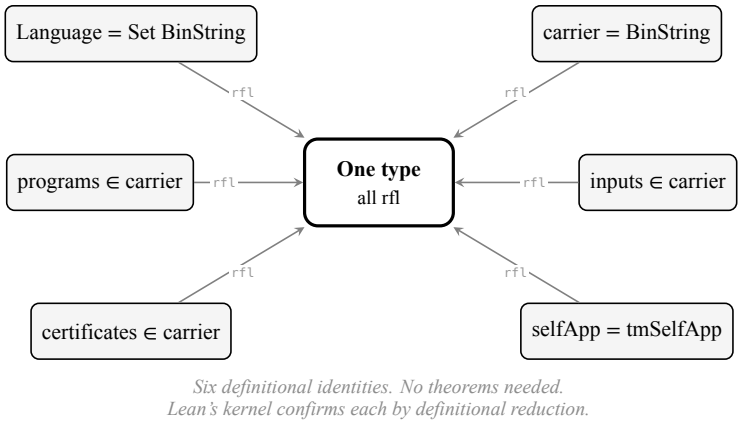


Figure 3: The rfl chain. Every object in the complexity-theoretic setup — languages, programs, inputs, certificates, selfApp — is definitionally identified with the carrier. These are not theorems; they are type equalities confirmed by Lean's kernel via `rfl`.

3.2 Two pairings, one carrier

Section 1/4 established: P = NP is a question about two pairings of computational model and language on a shared carrier — whether certificate access can always be eliminated. Section 1/2 established: this requires ReflData, and the classical carrier admits only WeakReflData. This section shows concretely what the compositions are.

Both pairings share the carrier (`BinString`), the encoding (fold/unfold), and the computation model (`evaln`). Programs, inputs, and certificates are all binary strings. The question is whether the asymmetry between the two pairings — verification has certificate access, decision does not — can be resolved on this carrier.

Every computation on the carrier passes through fold and unfold. The round trip through unfold then fold is selfApp. Section 1/2 established: the section does not close on the classical carrier (three independent proofs). The nonclosure constrains every computation on it.

What the compositions are, concretely: when two pairings share fold/unfold, the cross-pairing compositions collapse to the retraction and its dual (`GRM.toPNP_cross`, `Parallel.lean:133`):

```
GRM.toPNP_cross (M : GRM) (Lp Lnp : Set M.carrier) (x : M.carrier) :
    (M.toPNP Lp Lnp).p_eval_np x = x ∧            -- cross retract (roundtrip)
    (M.toPNP Lp Lnp).np_eval_p x = x ∧            -- cross retract (roundtrip)
    (M.toPNP Lp Lnp).p_abs_np x = M.selfApp x ∧    -- cross section: selfApp
    (M.toPNP Lp Lnp).np_abs_p x = M.selfApp x :=   -- cross section: selfApp
  ⟨M.roundtrip x, M.roundtrip x, rfl, rfl⟩          -- Parallel.lean:133
```

The cross-retract compositions are identity (the roundtrip, always). The cross-section compositions are selfApp (not necessarily identity). Both are unconditional — they hold for any two languages on any carrier with shared fold/unfold. This is not a modeling choice. It is what the compositions are (`rfl`).

The section closing is a single condition. selfApp = id forces both compositions to be identity — the retraction upgrades to a full isomorphism (`regime_determines_answer`, `Cascade.lean:182`). On the classical carrier, selfApp = id forces headerLen = 0, since selfApp([]) = header and identity requires header = []. headerLen = 0 yields a `FoldUnfoldSection` from $T(M)$ to M (`selfApp_id_gives_section`, `Parallel.lean:175`). The strict naming layer (drift = 0) exists. No certify/extract asymmetry is possible (`no_asymmetry_of_-`

selfApp_eq_id, Cascade.lean:171). These conditions are biconditional — each forces all the others. Group C is the equivalence class.

Three independent results prove the section does not close on the classical carrier; a fourth — the counting gap — operates at a different level, establishing that the naming obstruction is intrinsic to the carrier's information geometry (Section 3.4). The nonclosure is encoding-invariant (regime_B_iff, ReencodingInvariance.lean:404).

3.3 P and NP — standard definitions

P and NP are defined on BinString = carrier (Language.lean:69–95). InP quantifies over a step-bounded machine, a program (BinString), inputs (BinString), and polynomial time bounds. InNP quantifies over a step-bounded machine, a verifier program (BinString), certificates (BinString), and polynomial bounds for both certificate length and verification time. Step-bounded computation uses Mathlib's Nat.Partrec.Code.evaln (Carneiro 2019) (StepBounded.lean:85): Goedel-numbered partial recursive codes implementing exactly the partial recursive functions. P ⊆ NP (P_always_sub_NP, Language.lean:104).

selfApp acts on every carrier element — deciders, verifiers, inputs, and certificates alike (selfApp_eq_tmSelfApp, rfl). The fold/unfold used by the computation model are definitionally the same fold/unfold constrained by classicalGRM_factorsThrough, fold_unfold_nonclosure, and construction_super_poly (rfl). Verification inputs pair(x, w) with $|x| \geq 1$ are not on the substrate: they are structurally displaced by selfApp (verification_inputs_not_fixed, CarrierConnection.lean:183).

3.4 Four structural results

Each is an unconditional structural fact about the classical carrier.

Three independently prove the section does not close:

- **classicalGRM_factorsThrough** (RegimeClassification.lean:21): selfApp factors through headerLen. Self-reference costs exactly headerLen bits. The naming layer at drift 0 does not exist.
- **fold_unfold_nonclosure** (Nonclosure.lean:27): no FoldUnfoldSection from $T(M)$ to the classical carrier. The structural return from the frictionless model is impossible. Six lines, omega.
- **selfApp ≠ id** (presemantic_not_group_C, Cascade.lean:276): the section does not close. Pre-semantic, uses no grade function.

A fourth result operates at a different level:

- **construction_super_poly** (CountingEngine.lean:128): the endomorphism count super-polynomially exceeds description capacity at every grade. This is carrier arithmetic — N_{End} and N_{Val} are defined in terms of 2^{g+1}, independent of selfApp. The gap holds regardless of regime. It establishes that the naming obstruction is intrinsic to the carrier's information geometry, not merely a regime artifact.

The conjunction (classical_carrier_structure, CarrierConnection.lean) bundles factorsThrough, nonclosure, and the counting gap for any positive-length header. classical_nonclosure (CarrierConnection.lean) instantiates at concreteHeader: no parameters, no hypotheses.

3.5 The substrate and its complement

On Fix(selfApp) — the header-prefixed strings — both compositions are identity: fold ∘ unfold = id (the roundtrip, which holds globally) and unfold ∘ fold = id (the cotrip, which holds on the substrate

only) (`substrate_full_iso`, `CarrierConnection.lean:100`). The substrate carries the zero-cost structure: both compositions are identity, the structural obstructions vanish.

On the complement — everything not header-prefixed — selfApp is nontrivial. One definitional equality captures the entire action:

```
selfApp_two_to_one : selfApp (b :: rest) = true :: rest := rfl
                                        -- CarrierConnection.lean:208
```

selfApp replaces the first bit with the header, regardless of what it was. At each grade $g \geq 1$, exactly 2^{g-1} carrier elements start with true (fixed by selfApp) and 2^{g-1} start with false (displaced). The substrate and its complement split the carrier in half at every positive grade. The boundary is maximally sharp.

3.6 Displacement of computational content

selfApp cannot distinguish acceptance from rejection. Both [false] and [] map to [true] — the same carrier element that encodes acceptance (`selfApp_reject_displaced`, `selfApp_empty_displaced`, both `rfl`). The acceptance token [true] is the unique grade-1 fixed point (`selfApp_accept_fixed`, `rfl`).

Verification inputs `pair(x, w)` with $|x| \geq 1$ are displaced: they start with false (from the length prefix), not the header (`verification_inputs_not_fixed`, `CarrierConnection.lean:183`). Programs shorter than headerLen are displaced regardless of what they compute (`short_programs_displaced`, `CarrierConnection.lean:198`). selfApp preserves grade at or above headerLen (`selfApp_grade_eq_above`); below, it increases grade to exactly headerLen (`selfApp_grade_increases_empty`).

The structural results (Section 3.4) are proved on the full carrier. The displacement theorems show where the action is: the computationally meaningful elements — verification inputs, short programs, the reject token — are on the side of the carrier where selfApp is nontrivial.

3.7 The argument

The obstruction is a property of the carrier. The structural work is done before the complexity classes enter.

1. **The retraction equation forces the carrier architecture.** selfApp idempotent, Im(selfApp) = Fix(selfApp), witness search, finite type collapse. The architecture is the canonical splitting of the idempotent selfApp, unique up to unique isomorphism. (Tiers 0–2. Proved.)

2. **P is defined on this carrier.** Language = Set carrier. Programs and inputs are carrier elements. The computation model operates on carrier elements. selfApp = tmSelfApp. Each is `rfl`. (Tier 7. Definitional.)

3. **The section does not close on the classical carrier.** selfApp ≠ id — pre-semantic (`presemantic_not_group_C`). No structural section from the frictionless model (`fold_unfold_nonclosure`) — pre-semantic, six lines, `omega`. selfApp factors through headerLen — irreducible cost, tight (`classicalGRM_factorsThrough`). Three independent proofs. The super-polynomial counting gap (`construction_super_poly`) establishes that the naming obstruction is intrinsic to the carrier's information geometry. (Tiers 5–6. Proved.)

4. **Group C is the regime where the section closes.** selfApp = id gives: both compositions identity, strict naming layer, no certify/extract asymmetry, `FoldUnfoldSection` from $T(M)$ to M (`regime_determines_answer`, `no_asymmetry_of_selfApp_eq_id`, `selfApp_id_gives_section`). These are biconditional. (Tiers 0–4. Proved.)

5. **The obstruction is encoding-invariant.** Regime B is invariant under `BoundedGRMEquiv` (`regime_B_iff`). `AdmissibleEncoding` shows the classical carrier satisfies the encoding interface. `SameSemantics` preserves the regime across admissible encodings. The classification is a property of the carrier geometry, not the encoding. (Tier 2. Proved.)
6. **Nonclosure applies.** The section does not close on this carrier (Section 1, three independent proofs). P and NP are on this carrier (`rfl`). There is one type. P $\subseteq$ NP (`P_always_sub_NP`).

Every step is either `rfl` or a proved theorem. `structural_bridge` (`Parallel.lean:220`) bundles the rfl chain and the structural constraints. `displacement_witnesses` (`Parallel.lean:253`) shows concretely where selfApp acts on computational objects.

The Framework

The reader has seen the result, the proof, and why this arithmetic is about P vs NP. The remaining sections build the framework underneath: the universal theory (any GRM, any carrier), then the classical determination (binary strings with drop/prepend).

4. The One Equation

A *retraction* consists of two types X and Y, two morphisms $\alpha : X \to Y$ and $\beta : Y \to X$, and one equation:

$$\alpha(\beta(y)) = y \quad \text{for all } y \in Y.$$

(`Retraction`, `Primitives.lean:32`)

The dual composition $\sigma = \beta \circ \alpha$ is an endomorphism on X. From this single equation:

Idempotence. $\sigma(\sigma(x)) = \sigma(x)$ for all x. One rewrite from α_β (`Retraction.σ_idempotent`, `Primitives.lean:45`). Axioms: none.

Absorption. $\alpha \circ \sigma = \alpha$ (`Retraction.α_σ`, `Primitives.lean:52`). Axioms: none.

Witness search. If P is σ-preserved and some x satisfies P, a fixed point of σ satisfies P (`Retraction.witness_search`, `Primitives.lean:67`). Axioms: none.

Finite type collapse. On finite types, $\sigma = \mathrm{id}$ (`Retraction.fin_σ_eq_id`, `Primitives.lean:76`). Nontrivial regimes require infinite carriers.

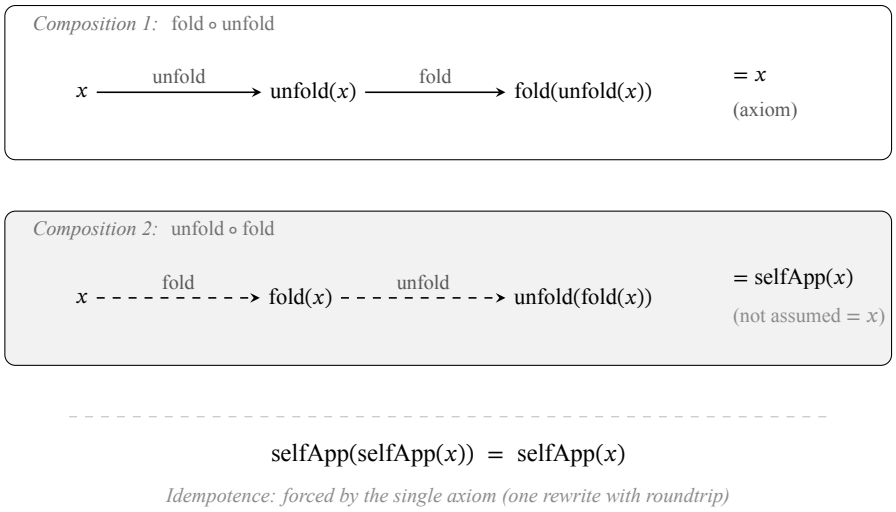


Figure 4: The retraction asymmetry. Composition 1 is the axiom; Composition 2 defines selfApp, which need not be the identity. Idempotence follows in one rewrite.

A *Graded Reflexive Model* (GRM) is the single-carrier specialization: $X = Y =$ carrier, $\alpha =$ fold, $\beta =$ unfold, with a grade function (`GRM`, `Primitives.lean:190`). The dual composition becomes selfApp = unfold ∘ fold.

The equation admits three structural positions, all inhabited under the same retraction axiom (`regime_conservativity`, `Cascade.lean:83`):

- **Group C**: selfApp = id. Both compositions identity. Witness: `trivialModel` (`group_C_inhabited`, `Regime.lean:113`).
- **Group B**: selfApp bounded, not identity. SelfAppBounded holds, selfApp ≠ id. Witness: `retractionModel` (`group_B_inhabited`, `Regime.lean:126`).
- **Separation**: selfApp unbounded. SelfAppUnbounded. Witness: `standardModel` (`separation_inhabited`, `Regime.lean:142`).

The classification is irrefutable in double-negation form (`classification_dn`, `Regime.lean:46`). The equation does not choose a regime. It classifies.

Preview. When the carrier is binary strings, fold = drop the header, unfold = prepend it, selfApp = the Kleene fixed-point construction (Kleene 1938). Header length > 0 is forced by the recursion theorem. But the abstract equation knows none of this.

5. The Carrier Architecture

Every GRM carries a canonical decomposition into Fix(selfApp) and its complement (`GRM.toReflectiveCarrierData`, `CarrierArchitecture.lean:186`). The canonicalizer is selfApp — definitionally (`GRM.reflectiveCarrierData_canonicalize`, `CarrierArchitecture.lean:200`, proved by `rfl`). The architecture requires no grade hypothesis.

The naming layer. Grade enters through inhabitation: the naming layer exists iff selfApp is grade-non-increasing (`GRM.grade_compatible_extension_iff`, `CarrierArchitecture.lean:266`). At drift k: iff selfApp increases grade by at most k (`GRM.drift_extension_iff`, `CarrierArchitecture.lean:315`). When the layer exists, it is unique — `Subsingleton` (`FixedGradeReflectiveCarrier.instSubsingleton`, `CarrierArchitecture.lean:227`; `FixedGradeDriftCarrier.instSubsingleton`, `CarrierArchitecture.lean:236`). The naming

convention is forced, not chosen.

Split idempotent. The carrier architecture is the canonical splitting of the idempotent selfApp. `SplitIdempotent` ↔ `ReflectiveCarrierData` with roundtrips preserving the endomorphism (`ReflectiveCarrierData.splitIdempotent_roundtrip`, `SplitIdempotent.lean:94`; `SplitIdempotent.reflectiveCarrierData_roundtrip`, `SplitIdempotent.lean:107`; both `rfl`). This is the standard categorical construction, not a bespoke one.

Universal property. Any compatible splitting of selfApp is canonically isomorphic to Fix(selfApp). The comparison map is the unique such isomorphism (`CompatibleSplitting.toFixed_unique`, `UniversalProperty.lean:163`; `CompatibleSplitting.fromFixed_unique`, `UniversalProperty.lean:174`). "Forced, not chosen" is a proved theorem.

Encoding invariance. A `BoundedGRMEquiv` is a bijection between two GRMs with bounded additive grade distortion, commuting with selfApp (`ReencodingInvariance.lean:48`). Under any such equivalence: `UnboundedGap` is invariant (`unboundedGap_iff`, `ReencodingInvariance.lean:211`). Finite drift existence is invariant (`finite_drift_iff`, `ReencodingInvariance.lean:241`). Regime B is invariant (`regime_B_iff`, `ReencodingInvariance.lean:404`). `BoundedGRMEquiv` forms a groupoid: `refl` (overhead 0), `symm` (same overhead), `trans` (additive overhead). Invariance composes. No bounded reencoding can collapse an infinite naming obstruction into a finite one. The regime classification is a property of the carrier geometry, not the encoding. An `AdmissibleEncoding` is a GRM with a specific overhead bound on selfApp (`AdmissibleEncoding.lean:39`). Two admissible encodings with `SameSemantics` — bounded bijective translations commuting with selfApp — yield a `BoundedGRMEquiv` (`SameSemantics.toBoundedGRMEquiv`, `AdmissibleEncoding.lean`). The `classicalAdmissibleEncoding` has overhead = headerLen.

Transport. For any GRM M, the transport model $T(M)$ is always Group C: selfApp = id (`transport_selfApp_eq_id`, `Transport.lean:117`). The meta-level is always frictionless. A `FoldUnfoldSection` from M_1 to M_2 is the weakest structural map: commutes with fold and unfold, no grade condition (`FoldUnfoldSection`, `Transport.lean:68`).

The regime determines the answer. selfApp = id implies: full isomorphism (both compositions identity), strict naming layer exists, no certify/extract asymmetry possible (`regime_determines_answer`, `Cascade.lean:182`; `no_asymmetry_of_selfApp_eq_id`, `Cascade.lean:171`). These are proved consequences.

Full development of Tiers 0–4 appears in Appendix B.

---

6. The Classical Carrier (Pre-Semantic)

The classical carrier is finite binary strings:

```
abbrev BinString := List Bool
```

(`BinString.lean:22`)

The operations: fold = tmFold: drop the first `headerLen` bits (`BinString.lean:62`). unfold = tmUnfold: prepend the header (`BinString.lean:66`). The retraction equation holds: fold(unfold(x)) = x (`tmFold_tmUnfold`, `BinString.lean:71`). The `classicalGRM` bundles carrier, fold, unfold, and grade = string length (`Instantiation.lean:30`). The pre-semantic results below (Sections 6.1–6.3) use only the fold/unfold geometry, not the grade function.

The carrier geometry is asymmetric: fold shortens by headerLen, unfold lengthens by headerLen. When headerLen > 0, this asymmetry is irreducible.

The theorems are universally quantified over `h : SelfAppHeader`. `classical_nonclosure` eliminates the parameter entirely at `concreteHeader = [true]`: no parameters, no hypotheses. `classical_carrier_structure` holds unconditionally for all positive-length headers.

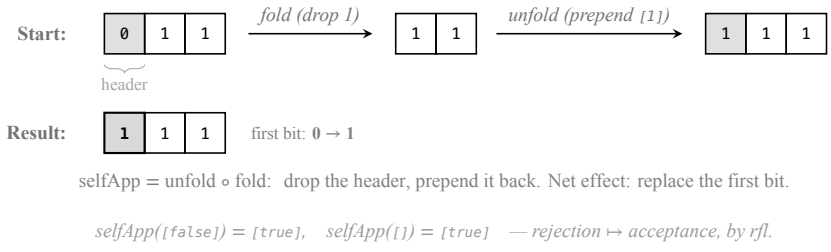


Figure 5: Concrete example: selfApp on `[false, true, true]`. Fold drops the first bit; unfold prepends `[true]`. Net effect: replace the first bit. selfApp maps both rejection and the empty string to acceptance — by rfl.

6.1 Structural nonclosure (pre-semantic)

This is the strongest single result. It uses no grade function, no counting argument, no complexity-theoretic vocabulary. Pure carrier geometry.

Theorem (`fold_unfold_nonclosure`, `Nonclosure.lean:27`). *For any* `SelfAppHeader` h *with* h.headerLen > 0, *there is no* `FoldUnfoldSection` *from* transportRM(classicalRM(h)) *to* classicalRM(h).

The type signature is pre-semantic: `classicalRM`, not `classicalGRM`. No grade function appears. The proof:

```
theorem fold_unfold_nonclosure (hne : h.headerLen > 0)
    (φ : FoldUnfoldSection (transportRM (classicalRM h))
                           (classicalRM h)) :
    False := by
  set R := classicalRM h
  set TR := transportRM R
  set t := Morphism.identity R
  have h_fold_id : TR.fold t = t := TR.roundtrip t
  have h1 : R.fold (φ.map t) = φ.map t := by
    have := φ.map_fold t; rw [h_fold_id] at this; exact this.symm
  have h2 : R.unfold (φ.map t) = φ.map t := by
    have := φ.map_unfold t; exact this.symm
  have h_eq : R.fold (φ.map t) = R.unfold (φ.map t) := by rw [h1, h2]
  have h_len : List.length (R.fold (φ.map t)) ≤ List.length (φ.map t) := by
    simp [R, classicalRM, tmFold, List.length_drop]
  have h_len2 : List.length (R.unfold (φ.map t)) ≥
      List.length (φ.map t) + h.headerLen := by
    simp [R, classicalRM, tmUnfold, List.length_append, h.headerLen_eq]; omega
  rw [h_eq] at h_len; omega
```

Let φ be such a section and t the identity morphism. In $T(R)$, fold fixes t (the roundtrip fires inside the transport). By commutation with fold: $R.\text{fold}(\varphi(t)) = \varphi(t)$. By commutation with unfold: $R.\text{unfold}(\varphi(t)) = \varphi(t)$. Therefore fold and unfold agree on $\varphi(t)$. But fold shortens by headerLen and unfold lengthens by headerLen. When headerLen > 0, this is a contradiction on lengths, closed by `omega`.

Six lines. The obstruction is geometric — fold shortens, unfold lengthens — and closes by arithmetic on list lengths (`omega`). The structural return from the frictionless model to the classical carrier is not difficult, not expensive. Impossible.

6.2 The section does not close (pre-semantic)

Theorem (`presemantic_not_group_C`, `Cascade.lean:276`). *If* headerLen > 0, *then* selfApp ≠ id.

selfApp([]) = header ++ [].drop(headerLen) = header. If selfApp were identity, then header = [], so headerLen = 0. Contradiction.

6.3 The substrate

Fix(selfApp) on the classical carrier: a binary string is fixed by selfApp iff it starts with the header (`selfApp_fixed_iff_header_prefix`, `CarrierConnection.lean:64`). Elements shorter than headerLen are displaced (`short_elements_not_fixed`, `CarrierConnection.lean:85`).

7. The Classical Carrier (Post-Semantic)

Grade = string length is now determined. The regime collapses to B.

7.1 selfApp factors through headerLen

Theorem (`classicalGRM_factorsThrough`, `RegimeClassification.lean:21`). *selfApp factors through headerLen.*

The bound is tight: selfApp on the empty string yields the header, whose grade is exactly headerLen (`classicalGRM_drift_exact`, `CarrierCounting.lean:151`). The minimum admissible drift is exactly headerLen (`classicalGRM_minimum_drift`, `CarrierCounting.lean:79`): for any $k <$ headerLen, the drifted naming layer at k does not exist. Not a loose bound. The exact irreducible cost.

7.2 The counting gap

$N_{\text{Val}}(g) = 2^{g+1}$: carrier elements at grade $\leq g$ (`CountingEngine.lean:23`). $N_{\text{End}}(g) = N_{\text{Val}}(g)^{N_{\text{Val}}(g)}$: endomorphisms on grade-$\leq g$ strings (`CountingEngine.lean:26`).

Theorem (`construction_super_poly`, `CountingEngine.lean:128`). *For every polynomial bound p,* ¬PolyBoundedConstruction($N_{\text{End}}, N_{\text{Val}}, p$).

$N_{\text{End}}(g) = (2^{g+1})^{2^{g+1}}$ is a tower. $N_{\text{Val}}(g + p(g)) = 2^{g+p(g)+1}$ is polynomial overhead into exponential. The tower outgrows any polynomial offset for all sufficiently large g (`growth_gap_survives_poly`, `CountingEngine.lean:113`).

This is an information-theoretic gap about describability, not computability. N_{Val} counts carrier elements — the programs, inputs, and certificates that P and NP quantify over. N_{End} counts behaviors on those elements. The carrier cannot polynomially describe its own behavioral richness at any grade.

7.3 The two-level naming gap

Theorem (`classicalGRM_two_level_naming_gap`, `CarrierCounting.lean:187`). When headerLen > 0: (1) element naming succeeds at drift = headerLen; (2) strict naming fails (drift = 0 uninhabitable); (3) function

naming fails at every polynomial.

Same carrier, two levels, opposite answers. Elements can be named with bounded overhead. Behaviors cannot.

7.4 Regime B confirmed

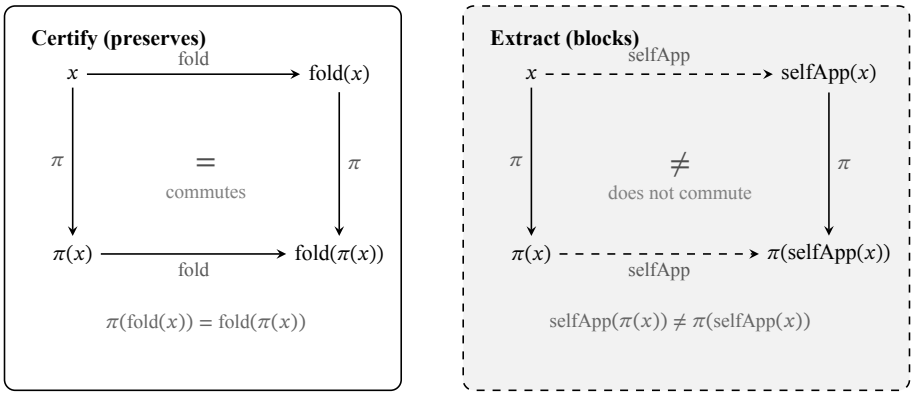
classicalGRM satisfies SelfAppBounded at d = headerLen (`classicalGRM_SelfAppBounded`, `RegimeClassification.lean:29`). SelfAppUnbounded does not hold (`classicalGRM_not_selfAppUnbounded`, `RegimeClassification.lean:33`). The section does not close (pre-semantically established, Section 6.2).

Naming note. `SelfAppBounded` is a regime classifier: $\exists d$, FactorsThrough(M, selfApp, d). It means selfApp has a finite grade bound. classicalGRM satisfies this — it is in Group B (bounded drift, not identity). This is not a contradiction with P ≠ NP. Group B is where selfApp ≠ id with finite cost. The structural results are about why selfApp ≠ id, not about whether selfApp is bounded.

7.5 Bounded certify/extract asymmetry

fold commutes with the zero projection at zero cost. selfApp does not: selfApp([]) = header ≠ [] (`classicalGRM_bounded_certify_extract`, `CarrierCounting.lean:134`). The cost is exactly headerLen — achieved, not just bounded.

All definitions computable. No `noncomputable` flags. Carrier, fold, unfold, grade are concrete operations.



The same retraction/section boundary at the transport level:
certification (fold direction) preserves; extraction (selfApp direction) blocks.

Figure 6: The certify/extract asymmetry. Fold commutes with projection π (left square); selfApp does not (right square). Certification preserves structure across the transport; extraction is blocked by the same retraction boundary.

8. Complexity on the Classical Carrier

The nonclosure is a property of the carrier. P is defined on this carrier (`rfl`). The section does not close (Sections 6–7). The nonclosure is encoding-invariant. NP is on the same carrier (`rfl`). P ⊆ NP (`P_always_sub_NP`). The proofs apply at each instance.

`structural_bridge` (`Parallel.lean:220`) bundles the rfl chain and the structural constraints as a single theorem for any header with positive length:

- **Identity**: Language = Set carrier, selfApp = tmSelfApp, TM operates on carrier. Each is `rfl`.
- **Structure**: on any GRM with shared fold/unfold, the cross-pairing compositions between two language pairings equal selfApp (`GRM.toPNP_cross`, `rfl`).
- **Constraint (nonclosure)**: selfApp $\neq$ id, no structural section, factorsThrough — three independent proofs that the section does not close. **Constraint (reinforcement)**: super-polynomial counting gap (intrinsic to carrier information geometry), bounded certify/extract asymmetry, P $\subseteq$ NP. Each is a proved theorem.

The obstruction is a carrier-geometric fact: the classical carrier is not in the regime where both compositions are identity. `AdmissibleEncoding` shows this is not an artifact of a special presentation — the classical carrier satisfies the encoding interface, and any same-semantics admissible encoding preserves the regime under bounded reencoding (`regime_B_iff`, `ReencodingInvariance.lean:404`).

`displacement_witnesses` (`Parallel.lean:253`) instantiates at `concreteHeader`: selfApp conflates [false] and [] to [true] (`rfl`), displaces verification inputs, displaces the empty string to the acceptance token (`rfl`). Axiom profile: `propext` only.

Non-vacuity. The carrier is inhabited ([] : BinString). The transport model is inhabited (`Transport.identity`). `PolyBound` is inhabited ($\langle 0, 0 \rangle$). No vacuous truth.

9. The Answer Space

9.1 One geometric parameter

The classical regime is fully determined by headerLen (`classical_regime_by_header`, `Cascade.lean:440`):

- headerLen = 0: selfApp = id, Group C, no obstructions (`zero_header_no_obstruction`, `Openness.lean:24`).
- headerLen > 0: `classical_carrier_structure` holds (`CarrierConnection.lean`).

The regime is a function of a single geometric parameter.

9.2 Four results from one geometric asymmetry

The four structural results (Section 3.4) are projections of one geometric fact: fold shortens, unfold lengthens, headerLen > 0. Three independently prove nonclosure. `fold_unfold_nonclosure` (pre-semantic, six lines, `omega`) is the sharpest. `classicalGRM_factorsThrough` (quantitative, tight drift bound) is the most informative. `presemantic_not_group_C` is the most direct: selfApp $\neq$ id from the carrier geometry alone. `construction_super_poly` (super-polynomial counting gap) operates at a different level — it establishes that the naming obstruction is intrinsic to the carrier's information geometry, not merely a regime artifact.

9.3 Openness

Group C is inhabited (`group_C_inhabited`, `Regime.lean:113`). The obstruction is specific to the classical geometry (positive header on binary strings with drop/prepend), not to computation itself. Non-classical geometries are not constrained by this result.

The regime classification is a theorem about retractions, not a framework designed to prove P $\neq$ NP. It classifies ALL retractions. Classical computation, given the recursion-theorem-forced positive header, occupies Group B. The position has consequences. This is measurement, not construction.

10. On the Standard Barriers

The three standard barrier results — relativization (Baker et al. 1975), natural proofs (Razborov and Rudich 1997), algebrization (Aaronson and Wigderson 2009) — each constrain proof techniques formulated within the complexity-theoretic framework. Relativization constrains diagonalization against oracles. Natural proofs constrains arguments from combinatorial properties of Boolean functions. Algebrization constrains algebraic extensions of oracle complexity. Each presupposes a setting in which P and NP are defined as complexity classes, and constrains which proof strategies within that setting can succeed.

The structural results here are formulated below that setting. The nonclosure theorem (`fold_unfold_nonclosure`) is arithmetic on list lengths under drop and prepend, closed by `omega`. It uses no oracle model, no Boolean function, no algebraic structure. The carrier's encoding geometry is fixed before P and NP are introduced. The barrier conditions are not formulated at this level.

The oracle case is addressed directly. An oracle does not change the carrier's encoding structure: `OracleAugmentedRM` is definitionally equal to the base model (`Cascade.lean:498`). `fold_unfold_nonclosure` holds regardless of oracle access (`fold_unfold_nonclosure_oracle_independent`, `Cascade.lean:502`). The nonclosure is a property of drop and prepend on binary strings, not of the computation model layered on top.

11. Related Work

Scott's reflexive domains. Scott's D_∞ construction (Scott 1972) builds reflexive domains where function spaces embed into the domain — an embedding-projection pair satisfying the retraction equation. $T(M)$ shares this structure: it is a reflexive object over M with selfApp = id. The GRM framework differs in classifying carriers by the *obstruction* to achieving the reflexive property, rather than constructing domains where it holds.

The lambda calculus. The lambda calculus (Barendregt 1984) requires a reflexive domain for self-application. The GRM's retraction fragment (`Retraction`) is the minimal structure: one equation, not two. The gap between one equation (retraction, selfApp idempotent) and two (full isomorphism, selfApp = id) is the regime classification.

The recursion theorem. The self-application header is the Kleene fixed-point construction (Kleene 1938) on binary strings. Header length is the irreducible cost. The recursion theorem guarantees positive length. The GRM framework measures the structural consequences.

Idempotent splitting. The splitting of an idempotent endomorphism into a retraction is the Karoubi envelope (Karoubi 1978) (idempotent completion) of a category. The carrier architecture is an instance: selfApp is idempotent, and its splitting into Fix(selfApp) and complement is the standard construction. The contribution is not the splitting itself but the universal property proved for the specific splitting that computation forces, and the encoding invariance that makes the regime classification a geometric property rather than an encoding artifact.

Barrier results. Baker et al. (1975) showed relativizing techniques cannot resolve P vs NP. Razborov and Rudich (1997) showed natural combinatorial properties of Boolean functions face a cryptographic obstacle. Aaronson and Wigderson (2009) showed algebrizing techniques face a similar barrier. These results constrain proof techniques formulated within complexity theory. The structural results here operate at the carrier's encoding geometry level — list lengths under drop and prepend — where the barrier conditions have not been formulated (Section 10). This is determination-level priority, not evasion.

Formal verification. Machine-checked complexity theory is developing: undecidability results in Coq

(Forster et al. 2019), computability in Lean and Mathlib (Carneiro 2019), complexity theory in Coq (Gäher and Kunze 2021). Existing formalizations typically mechanize known results. This development differs in that the Lean project is the proof — the primary artifact, not a transcription of a pen-and-paper argument.

Prior approaches to P vs NP. The P vs NP problem, first formally posed by Cook (1971), has been approached via diagonalization, circuit complexity, algebraic geometry, and proof complexity — techniques that operate on specific computational models or proof systems. The GRM framework characterizes the carrier's encoding structure. The structural results are properties of binary strings under drop/prepend, not properties of Turing machines, circuits, or algebraic proof systems. The barriers constrain techniques inside the complexity-theoretic framework. The structural results are about the encoding geometry that underlies it.

Appendix A: The Lean Formalization

A.1 Project structure

Core/ (9 files, Tiers 0–4): the universal framework.

File	Lines	Role
`Primitives.lean`	292	Retraction, GRM, FullRetraction
`Regime.lean`	145	Regime predicates, classification, concrete models
`Transport.lean`	141	Transport model $T(M)$, FoldUnfoldSection
`CarrierArchitecture.lean`	453	Carrier decomposition, naming layer, overflow confinement
`CertifyExtract.lean`	223	Projection, certify/extract asymmetry
`SplitIdempotent.lean`	188	Categorical splitting; ReflectiveCarrierData ↔ SplitIdempotent
`UniversalProperty.lean`	341	Universal property; architecture unique up to unique iso
`ReencodingInvariance.lean`	413	BoundedGRMEquiv; encoding invariance; regime B invariance
`AdmissibleEncoding.lean`	80	AdmissibleEncoding, SameSemantics → BoundedGRMEquiv

Classical/ (7 files, Tiers 5–8): the classical determination.

File	Lines	Role
`BinString.lean`	86	Binary string carrier, fold/unfold geometry
`Instantiation.lean`	57	classicalGRM construction
`Nonclosure.lean`	54	Structural nonclosure
`RegimeClassification.lean`	41	selfApp factors through, Regime B
`CountingEngine.lean`	134	Super-polynomial counting gap
`Openness.lean`	32	headerLen = 0 gives Group C
`CarrierCounting.lean`	198	Two-level naming gap, bounded asymmetry

Two files removed during refactor: `ThreeCannots.lean` (conjunction moved to `CarrierConnection.lean`) and `BarrierEvasion.lean` (determination-level priority, see Section 10).

Classical/Complexity/ (4 files, Tier 7): complexity on the carrier.

File	Lines	Role
`StepBounded.lean`	189	Step-bounded TM, pair encoding
`Language.lean`	135	P, NP, P ⊆ NP
`CarrierConnection.lean`	240	Carrier identity, displacement, structural properties on complexity carrier
`Parallel.lean`	281	PNP structure, cross-pairing, structural_bridge, displacement_witnesses

Root:

File	Lines	Role
`Cascade.lean`	532	Determination hierarchy, oracle independence, axiom audit
`OneEquation.lean`	27	Root import aggregator

(21 source files, 1 root import aggregator.)

A.2 Axiom audit

Every theorem uses only standard Lean 4 kernel axioms. Representative profiles:

Theorem	Axioms
`Retraction.σ_idempotent`	none
`GRM.selfApp_idempotent`	none
`group_C_inhabited`	none
`GRM.grade_compatible_extension_iff`	`propext`, `Quot.sound`
`transport_selfApp_eq_id`	`Quot.sound`
`no_asymmetry_of_selfApp_eq_id`	none
`fold_unfold_nonclosure`	`propext`, `Classical.choice`, `Quot.sound`
`classicalGRM_factorsThrough`	`propext`, `Quot.sound`
`construction_super_poly`	`propext`, `Classical.choice`, `Quot.sound`
`P_always_sub_NP`	`propext`, `Classical.choice`, `Quot.sound`
`complexity_on_classical_carrier`	`propext`, `Classical.choice`, `Quot.sound`
`selfApp_id_gives_section`	`propext`, `Classical.choice`, `Quot.sound`

These are the transitive Lean kernel dependencies reported by `#print axioms` — standard Lean 4/Mathlib foundations. Zero custom axioms.

A.3 Build instructions

```
lake build
```

Lean 4 v4.28.0, Mathlib v4.28.0. Zero `sorry` verified by grep. Axiom profiles verified by `#print axioms` in `Cascade.lean` (lines 518–530).

A.4 Computational soundness

The computation chain: $\text{BinString} \xrightarrow{\text{bitsToNat}} \mathbb{N} \xrightarrow{\text{ofNatCode}} \text{Code} \xrightarrow{\text{evaln}} \text{Option}\ \mathbb{N} \xrightarrow{\text{natToBits}} \text{BinString}$. The structural argument does not depend on encoding roundtrip faithfulness — it concerns the carrier's fold/unfold geometry.

Appendix B: The Determination Cascade (Formal)

The complete tier structure as formalized in `Cascade.lean`:

Tier 0: Retraction. Two types, two morphisms, one equation: $\alpha \circ \beta = \text{id}$. Forces: σ idempotent, $\text{Im}(\sigma) = \text{Fix}(\sigma)$, witness search, finite type collapse. Free: the regime.

Tier 1: GRM + Regime. Single-carrier specialization. selfApp = unfold ∘ fold. Three regime positions, all inhabited. The retraction axiom does not choose.

Tier 2: Carrier Architecture. The forced decomposition into Fix(selfApp) and complement. canonicalize = selfApp by `rfl`. Naming layer inhabitation is the regime question (iff-characterization). When the layer exists: unique (Subsingleton). The drift spectrum: strict (drift 0), bounded-loss (drift k), unbounded gap (no finite drift). Admissible drift set upward-closed (`GRM.drift_extension_mono`, `CarrierArchitecture.lean:354`). Full biconditionals: `selfAppUnbounded_iff_not_SelfAppBounded` (`Regime.lean:69`), `unbounded_gap_iff_no_finite_drift` (`CarrierArchitecture.lean:364`).

The architecture is the canonical splitting of the idempotent selfApp (`SplitIdempotent.lean`). The splitting is unique up to unique isomorphism: any compatible splitting of selfApp is canonically isomorphic to Fix(selfApp), and the comparison map is the unique compatible isomorphism (`CompatibleSplitting.toFixed_unique`, `UniversalProperty.lean:163`).

Encoding invariance: `BoundedGRMEquiv` preserves `UnboundedGap` (`unboundedGap_iff`, `ReencodingInvariance.lean:211`), finite drift existence (`finite_drift_iff`, `ReencodingInvariance.lean:241`), and Regime B (`regime_B_iff`, `ReencodingInvariance.lean:404`). Fixed-point subdomains transfer: `Fix(selfApp_M) \cong Fix(selfApp_{M'})` with bounded grade distortion (`fFixed`/`gFixed`, `ReencodingInvariance.lean:266--287`).

Image confinement: overflow witnesses are confined to $A \setminus \text{Fix}(\text{selfApp})$. Fixed points are immune to overflow (`CarrierArchitecture.lean`).

Tier 3: Transport. $T(M)$ always Group C (`transport_selfApp_eq_id`, `Transport.lean:117`). Admits strict naming layer (`transportGRM_admits_strict_extension`, `CarrierCounting.lean:102`). Upgrades to `FullRetraction` (`transportGRM_fullRetraction`, `CarrierCounting.lean:109`).

Tier 4: Certify/Extract Asymmetry. Verification (fold direction) commutes with projection (`compatible_preserves_certify`, `CertifyExtract.lean:70`). Extraction (selfApp direction) may not. Group C: no asymmetry (`no_asymmetry_of_selfApp_eq_id`, `Cascade.lean:171`). Group B: bounded asymmetry (`boundedCertifyExtractAsymmetry_of_witness`, `CertifyExtract.lean:140`). Separation: any projection blocks extraction (`certifyExtractAsymmetry_of_unbounded`, `CertifyExtract.lean:167`).

Tier 5: Pre-Semantic Classical. Carrier = BinString, fold = drop, unfold = prepend. `fold_unfold_nonclosure` applies (`Nonclosure.lean:27`). Not Group C (`presemantic_not_group_C`, `Cascade.lean:276`). Substrate characterized: Fix(selfApp) = header-prefixed strings (`selfApp_fixed_iff_header_prefix`, `CarrierConnection.lean:64`). Short elements displaced (`short_elements_not_fixed`, `CarrierConnection.lean:85`).

Tier 6: Post-Semantic Classical. Grade = length. `classicalGRM_factorsThrough`: drift = headerLen, tight (`classicalGRM_factorsThrough`, `RegimeClassification.lean:21`; `classicalGRM_drift_exact`, `CarrierCounting.lean:151`; `classicalGRM_minimum_drift`, `CarrierCounting.lean:79`). `construction_super_poly`: super-polynomial gap (`construction_super_poly`, `CountingEngine.lean:128`). Two-level naming gap (`classicalGRM_two_level_naming_gap`, `CarrierCounting.lean:187`). Regime B confirmed (`classicalGRM_SelfAppBounded`, `RegimeClassification.lean:29`).

Tier 7: Complexity on the Classical Carrier. Language = Set carrier (`rfl`). P, NP defined. $P \subseteq NP$. Cross-pairing composition = selfApp (`GRM.toPNP_cross`, `rfl`). `classical_carrier_structure` holds unconditionally on the complexity carrier (`classical_nonclosure`, `CarrierConnection.lean`). Substrate carries full isomorphism (`substrate_full_iso`). selfApp replaces the first bit: `selfApp_two_to_one` (`rfl`). Accept/reject conflation (`selfApp_reject_displaced`, `rfl`). Summary: `complexity_on_classical_carrier` (`CarrierConnection.lean:154`).

Tier 8: Answer Space. Regime by headerLen (`classical_regime_by_header`, `Cascade.lean:440`). Group C inhabited. Framework open.

Appendix C: Theorem-to-Claim Correspondence

Axiom key: P = `propext`, C = `Classical.choice`, Q = `Quot.sound`. All standard Lean 4/Mathlib. Complete theorem inventory (329 declarations, machine-verified axiom profiles) is in `CLAIMS.md` in the archived repository.

Lean Theorem	File:Line	Ax.	Role
`fold_unfold_nonclosure`	Nonclosure:27	PCQ	exclusion
`presemantic_not_group_C`	Cascade:276	P	exclusion
`classicalGRM_factorsThrough`	RegimeClass:21	PQ	exclusion
`construction_super_poly`	CountEngine:128	PCQ	reinforcement
`regime_determines_answer`	Cascade:182	PQ	biconditional
`no_asymmetry_of_selfApp_eq_id`	Cascade:171	—	biconditional
`classicalGRM_bounded_certify_extract`	CarrierCount:134	PCQ	constructed
`language_is_carrier_set`	CarrierConn:35	rfl	rfl
`selfApp_eq_tmSelfApp`	CarrierConn:53	rfl	rfl
`GRM.toPNP_cross`	Parallel:133	rfl	rfl
`P_always_sub_NP`	Language:104	PCQ	complexity
`structural_bridge`	Parallel:220	PCQ	bundle
`regime_B_iff`	ReencInv:404	PCQ	invariance
`fold_unfold_nonclosure_oracle_independent`	Cascade:502	PCQ	oracle

Aaronson, Scott, and Avi Wigderson. 2009. "Algebrization: A New Barrier in Complexity Theory." *ACM Transactions on Computation Theory* 1 (1). https://doi.org/10.1145/1490270.1490272.

Baker, Theodore, John Gill, and Robert Solovay. 1975. "Relativizations of the $\mathcal{P} = ?\mathcal{NP}$ Question." *SIAM Journal on Computing* 4 (4): 431–42. https://doi.org/10.1137/0204037.

Barendregt, Henk P. 1984. *The Lambda Calculus: Its Syntax and Semantics*. Revised. Vol. 103. Studies in Logic and the Foundations of Mathematics. North-Holland.

Carneiro, Mario. 2019. “Formalizing Computability Theory via Partial Recursive Functions.” *10th International Conference on Interactive Theorem Proving (ITP 2019)*, Leibniz international proceedings in informatics (LIPIcs), vol. 141: 12:1–17. https://doi.org/10.4230/LIPIcs.ITP.2019.12.

Close, Larsen James. 2026. *OneEquation: Structural Consequences of the Retraction Equation, Machine-Checked in Lean 4*. V. v1.0.0. Released. https://doi.org/10.5281/zenodo.19268096.

Cook, Stephen A. 1971. “The Complexity of Theorem-Proving Procedures.” *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC ’71)*, 151–58. https://doi.org/10.1145/800157.805047.

Forster, Yannick, Dominik Kirst, and Gert Smolka. 2019. “On Synthetic Undecidability in Coq, with an Application to the Entscheidungsproblem.” *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP ’19)*, 38–51. https://doi.org/10.1145/3293880.3294091.

Gäher, Lennard, and Fabian Kunze. 2021. “Mechanising Complexity Theory: The Cook–Levin Theorem in Coq.” *12th International Conference on Interactive Theorem Proving (ITP 2021)*, Leibniz international proceedings in informatics (LIPIcs), vol. 193: 20:1–18. https://doi.org/10.4230/LIPIcs.ITP.2021.20.

Karoubi, Max. 1978. *K-Theory: An Introduction*. Vol. 226. Grundlehren Der Mathematischen Wissenschaften. Springer. https://doi.org/10.1007/978-3-540-79890-3.

Kleene, Stephen C. 1938. “On Notation for Ordinal Numbers.” *Journal of Symbolic Logic* 3 (4): 150–55. https://doi.org/10.2307/2267778.

Moura, Leonardo de, and Sebastian Ullrich. 2021. “The Lean 4 Theorem Prover and Programming Language.” *Automated Deduction – CADE 28*, Lecture notes in computer science, vol. 12699: 625–35. https://doi.org/10.1007/978-3-030-79876-5_37.

Razborov, Alexander A., and Steven Rudich. 1997. “Natural Proofs.” *Journal of Computer and System Sciences* 55 (1): 24–35. https://doi.org/10.1006/jcss.1997.1494.

Scott, Dana. 1972. “Continuous Lattices.” In *Toposes, Algebraic Geometry and Logic*, edited by F. William Lawvere, vol. 274. Lecture Notes in Mathematics. Springer. https://doi.org/10.1007/BFb0073967.

The mathlib Community. 2020. “The Lean Mathematical Library.” *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP ’20)*, 367–81.