Execution Categories for Stochastic Program Generators

Quotient Semantics for Deterministic Executable Identity

Thomas Dionysopoulos, CFA

Abstract

Stochastic program generators produce syntactically diverse proposals for identical computations. Prior work established operational machinery for this setting: a grounding gate enforcing admissibility boundaries between AI proposals and formal execution [1], and canonical execution semantics providing deterministic replay identity via content-addressed hashing [2]. This paper formalizes the semantic structure those systems imply.

We define a registry-indexed execution category $\mathcal{B}_R$ whose objects are typed executable artifacts and whose morphisms are admissible pipeline transformations. The operational equivalence generated by the system's rewrite rules—alias resolution, argument-order normalization, canonical form selection—forms a congruence: equivalent subexpressions remain equivalent under arbitrary well-typed pipeline composition. The resulting quotient category $\mathcal{B}_R/\sim_R$ gives precise meaning to deterministic execution identity. Content-addressed hashing, applied to canonical representatives, serves as a computable operational witness of quotient membership under standard collision-resistance assumptions.

A projection $\pi : \mathcal{E}_R \to \mathcal{B}_R/\sim_R$ connects stochastic proposals to their execution classes, with fibers measuring collapse from surface diversity to canonical identity. The grounding gate restricts this projection to admissible proposals, and the empirical measurements from Papers 1–2—replay determinism across 50 runs, canonicalization collapse across 1,200 LLM generations, hash stability under registry evolution—become interpretable as structural properties of the quotient and its fibers.

All constructions are operational, registry-relative, and grounded in the empirical systems of Papers 1–2. The semantics do not claim universal program equivalence; they formalize exactly the invariances that a registry-indexed execution system declares and enforces.

1 Introduction

When a stochastic generator produces two syntactically different proposals that should yield the same computation, what makes them "the same"?

Prior work gave operational answers. The grounding gate [1] enforces an admissibility boundary: no AI-generated proposal reaches execution without evidence that every referenced capability exists in the target system. Canonical execution semantics [2] provides deterministic replay identity: content-addressed hashing assigns identical hashes to identical execution plans regardless of surface variation. Together, these systems empirically demonstrate that stochastic diversity in the proposal space collapses to deterministic identity in the execution space—across 1,200 LLM calls and 50 replay runs, the collapse is measurable and the identity is stable.

But those systems implicitly depend on a semantic structure they do not formalize. The grounding gate assumes an admissibility boundary that is well-behaved under composition. Content-addressed hashing assumes equivalence classes that are compositionally stable. The canonicalization pipeline assumes that local rewrites are safe in arbitrary pipeline contexts. None of these assumptions are proved in Papers 1–2; all are empirically observed.

This paper makes those assumptions explicit. We define a registry-indexed execution category $\mathcal{B}_R$ and establish operationally that the equivalence generated by canonicalization rules is a congruence—executable equivalence is stable under pipeline composition. This single property elevates canonicalization from local rewrite normalization to compositional execution semantics. The quotient category $\mathcal{B}_R/\sim_R$ models deterministic execution identity. A projection over this quotient connects stochastic proposals to their canonical execution classes, giving categorical form to the empirical collapse measured in Paper 2 [2].

Scope and non-claims. The semantics are deliberately constrained. They formalize exactly the invariances that the execution system declares—alias resolution, argument-order normalization, canonical form selection—and nothing more. They are registry-relative: the equivalence classes change when the capability registry changes. They do not claim universal program equivalence, denotational completeness, or anything about the meaning of programs beyond their operational identity under the system's declared rewrite rules.

The category theory in this paper is a tool, not the point. We use categories, morphisms, quotients, and projections because they give precise language to the compositional structure that the operational system exhibits. We do not use monads, adjunctions, toposes, or enriched categories. The audience is AI systems, programming languages, and execution semantics researchers, not category theorists.

Contributions.

1. A **registry-indexed execution category** $\mathcal{B}_R$ whose objects are typed executable artifacts and whose morphisms are admissible deterministic transformations (§2).
2. An **operational invariance algebra** $\sim_R$ generated by four classes of declared rewrite rules, shown to be operational, constrained, and registry-relative (§3).
3. An **operational congruence proposition**: $\sim_R$ is stable under well-typed pipeline composition (§4)—the central result.
4. A **quotient execution category** $\mathcal{B}_R/\sim_R$ with content-addressed hashing as a computable operational witness of quotient membership (§5).
5. A **projection with fibers** connecting the stochastic proposal space to the quotient, with the grounding gate interpreted as fiber restriction (§6).
6. Explicit **categorical reinterpretation** of all empirical results from Papers 1–2 (§9).

Notation. We suppress the registry subscript R when it is clear from context, writing $\mathcal{B}$ for $\mathcal{B}_R$ and $\sim$ for $\sim_R$. $\mathcal{H}$ denotes the hash space (SHA-256 digests). When the registry-relative nature of a statement matters, the subscript is restored.

2 Registry-Indexed Execution Categories

This section defines the execution category $\mathcal{B}_R$. The construction is grounded in the operational system described in Papers 1–2: objects correspond to the typed artifacts that the system manipulates, and morphisms correspond to the admissible transformations that the system applies.

2.1 Registry

Definition 2.1 (Registry)**.** A *registry* $R = (D, A, F, M)$ consists of:

- D: a *capability dictionary*—the finite set of accepted operation names with their metadata (arity, type signature, layer assignment, IR variant).
- A: an *alias table*—rewrite rules mapping surface names to canonical names (e.g., `cs1` $\mapsto$ `run_sum`).
- F: a *family registry*—strategy families with typed parameter schemas and signal-block definitions (e.g., `MOM_WZS`, `CARRY`).
- M: a *metric registry*—scoring metrics with computation rules (e.g., `SRP`, `MDD`, `VOL`).

R is finite, enumerable, and versioned. The registry state is witnessed by a content-addressed hash DIC_HSH(R). Changing any component of R changes this hash.

The registry is the parameter that makes the entire formalization relative. Every definition, every equivalence class, and every identity judgment that follows is indexed by R. When the registry changes, the execution category changes. This is not a defect of the formalism; it reflects the operational reality that the system's capabilities evolve. Paper 2 verified empirically that identical registries produce identical hashes across independent runs, and that registry extension is detectable through hash comparison.

2.2 Objects

Definition 2.2 (Execution category — objects)**.** The objects of $\mathcal{B}_R$ are typed executable artifacts admissible under R:

Type	Description	Paper 2 name
Spec_R	Typed specifications: goal, family, metric, parameters, sweep	FPR
Expr_R	Well-formed expressions composed from operations in D	Pipeline expressions
Plan_R	Execution plans: expanded, canonicalized pipeline instances with content-addressed identity	MOR
Result_R	Scored results, selected artifacts, execution outputs	SCR, FPT

Each object has a declared type. Morphisms between objects of incompatible types are not defined.

In the operational system, a specification `(FPR :GOAL SWEEP :FAM MOM_WZS :SCR SRP :PRM (TUP :SIG_WIN (list 25 50 100)))` is an object of type Spec_R. The expansion of that specification into three concrete morphisms—one per `SIG_WIN` value—produces three objects of type Plan_R. Each plan, when evaluated against data, produces a Result_R object.

2.3 Morphisms

Definition 2.3 (Execution category — morphisms)**.** A morphism $f : a \to b$ in $\mathcal{B}_R$ is an admissible deterministic transformation between typed execution objects. Morphisms are generated by:

- *Registered operations*: each operation op $\in D$ induces morphisms on expressions and data (e.g., `dlog` maps an expression to a compound expression, or a data frame to a transformed frame).
- *Pipeline composition*: if $f : a \to b$ and $g : b \to c$, then $g \circ f : a \to c$. Composition is associative.
- *Parameter binding*: instantiating a sweep variable to a concrete value, mapping a specification to a plan.
- *Expression construction*: building compound expressions from subexpressions (threading, nesting, sequencing).

Each object has an identity morphism id_a.

Example 2.4. The pipeline expression `(wzs (dlog x) 25 1)` is the composition $\texttt{wzs}_{25,1} \circ \texttt{dlog}$, a morphism from input data to a rolling z-score of log returns. The threading form `(-> x (dlog) (wzs 25 1))` denotes the same composition.

2.4 Distinguished System Maps

The following are distinguished deterministic maps that operate on or within $\mathcal{B}_R$. They are named system operations, not arbitrary morphisms. When their source and target types align, they may be represented as morphisms; the paper treats them as architecturally distinguished rather than blurring them into the general morphism structure.

Map	Signature	Role
ν_R	$\text{Expr}_R \to \text{Expr}_R$	*Normalization*: alias resolution, surface-form rewriting
κ_R	$\text{Expr}_R \to \text{Expr}_R$	*Canonicalization*: canonical representative selection
ξ_R	$\text{Spec}_R \to \mathcal{P}(\text{Plan}_R)$	*Expansion*: specification to morphism set (Cartesian product over sweep parameters)
ε_R	$\text{Plan}_R \times \text{Data} \to \text{Result}_R$	*Evaluation*: deterministic execution
H	$\text{Expr}_R \to \mathcal{H}$	*Hashing*: content-addressed hash computation

ν_R and κ_R are endomorphisms on Expr_R and compose as morphisms. ξ_R changes type (specification to set of plans) and is better understood as a system-level operation than as a single morphism. H maps out of the category to an external hash space.

The key compositional property—that local application of ν_R or κ_R within a pipeline produces globally equivalent results—is the congruence established in §4.

Connection to Paper 2. The pipeline model of Paper 2—$G \to \Gamma \to \kappa \to E \to H$—maps directly onto these distinguished maps. The stochastic generator G produces candidates; the grounding gate Γ (Paper 1) restricts to admissible proposals; κ_R selects canonical representatives; ε_R evaluates deterministically; H witnesses the result.

3 Operational Invariance Algebra

This section defines the equivalence relation $\sim_R$ on objects of $\mathcal{B}_R$. The equivalence is generated by the specific rewrite rules that the system declares. It is operational: defined by transformation rules, not by program behavior. It is constrained: it does not equate all programs that compute the same function. It is registry-relative: extending the alias table extends the equivalence.

Definition 3.1 (Operational equivalence)**.** The *operational equivalence* $\sim_R$ on objects of $\mathcal{B}_R$ is the reflexive, symmetric, transitive closure of the following declared invariances:

(E1) Alias equivalence. If the alias table A maps surface name n to canonical name n', then any expression containing n is equivalent to the expression with n replaced by n'.

Example. `(cs1 x)` $\sim_R$ `(run_sum x)`, because A contains the rule `cs1` $\mapsto$ `run_sum`.

(E2) Argument-order invariance. Keyword arguments in typed specifications are order-independent.

Example. `(FPR :FAM MOM_WZS :SCR SRP)` $\sim_R$ `(FPR :SCR SRP :FAM MOM_WZS)`.

(E3) Canonical rewrite rules. The rewrite rules declared by the canonicalization pass:

- Threading elimination: `(-> x (f) (g))` $\sim_R$ `(g (f x))`.
- Composite decomposition: `(diff x k)` $\sim_R$ `(sub x (shift x k))`.
- Declared operator fusion, when the system specifies two expressions as semantically equivalent under fusion rules.

(E4) Description/identity separation. Objects differing only in descriptive metadata (DISC fields) are equivalent. Only semantic (SEM), algebraic (ALG), and implementation (IMPL) fields contribute to identity.

Example. Adding a human-readable description to an operation, or enriching its alias list, does not change its equivalence class.

Remark 3.2 (Three critical properties of $\sim_R$).

1. **Operational.** $\sim_R$ is defined by declared rewrite rules, not by program behavior. Two programs that compute the same function on all inputs are *not* necessarily equivalent under $\sim_R$ unless the declared rules derive that equivalence.
2. **Constrained.** $\sim_R$ is conservative by design. It captures the invariances the system explicitly guarantees, not all invariances that happen to hold. Soundness (equivalent things are treated as the same) is prioritized over completeness (all same things are detected as equivalent).
3. **Registry-relative.** $\sim_R$ depends on R. Adding an alias to A extends $\sim_R$ by making new pairs of expressions equivalent. Removing an alias shrinks it. The equivalence classes are contingent on the registry state.

Example 3.3 (Non-example: behavioral equivalence without operational equivalence)**.** Consider two expressions that compute the same result on all inputs but use different internal algorithms (e.g., two different sorting strategies). Unless the system's rewrite rules explicitly declare them equivalent, they are *not* equivalent under $\sim_R$. They will have different canonical forms, different hashes, and different provenance records. This is intentional: the system only guarantees identity for equivalences it can verify through its declared transformations.

4 Congruence and Compositionality

This is the central section of the paper. We establish that the operational equivalence $\sim_R$ is a congruence on $\mathcal{B}_R$: equivalent subexpressions remain equivalent when embedded in larger pipeline compositions. This property is what makes canonicalization compositionally safe—a local rewrite cannot break a global pipeline.

4.1 The Congruence Property

Proposition 4.1 (Operational congruence)**.** *Under the declared rewrite system* (E1)–(E4) *and the typing discipline of $\mathcal{B}_R$, the operational equivalence $\sim_R$ is a congruence. That is, for all morphisms f, g, h, k in $\mathcal{B}_R$:*

$$f \sim_R g \implies h \circ f \circ k \sim_R h \circ g \circ k$$

whenever the compositions $h \circ f \circ k$ and $h \circ g \circ k$ are well-typed.

Establishment. We verify that each generator of $\sim_R$ is preserved under pre- and post-composition with arbitrary well-typed morphisms.

- **(E1) Alias equivalence is preserved.** Alias resolution is performed at parse time, before any pipeline operator evaluates. Pipeline operators receive resolved names and cannot observe whether an alias was present in the original surface form. Pre- and post-composition with any morphism therefore preserves alias equivalence.

  Concretely: if `(cs1 x)` $\sim_R$ `(run_sum x)` by alias resolution, then `(wzs (dlog (cs1 x)) 25 1)` $\sim_R$ `(wzs (dlog (run_sum x)) 25 1)`, because `wzs` and `dlog` operate on the resolved form and cannot distinguish the two inputs.

- **(E2) Argument-order invariance is preserved.** Pipeline operators access specification fields by name, not by argument position. Reordering keyword arguments produces an identical field map. Any downstream morphism that reads fields by name cannot distinguish the two orderings.

  Concretely: if a specification s is expanded via $\xi_R(s)$, the resulting morphism set depends on the field values, not on the syntactic order in which they appear in s.

- **(E3) Canonical rewrite rules are preserved.** Each rewrite rule in the canonicalization pass is context-free: it depends only on the local subexpression structure, not on the surrounding pipeline context. A rewrite that is valid in isolation remains valid when the subexpression is embedded in a larger composition.

  Concretely: the threading transformation `(-> x (f) (g))` $\sim_R$ `(g (f x))` is purely syntactic and does not depend on what f or g compute or what context they appear in. Post-composing with any morphism h preserves the equivalence because h receives the same value regardless of which syntactic form produced it.

- **(E4) Description/identity separation is preserved.** Descriptive metadata (DISC fields) does not participate in any computation or morphism in $\mathcal{B}_R$. No pipeline operator reads, transforms, or branches on DISC fields. Composition with any morphism therefore preserves description/identity equivalence.

Since each generator is preserved under composition, their reflexive, symmetric, transitive closure $\sim_R$ is also preserved under composition. □

Remark 4.2 (Scope of the congruence). This result depends on specific structural properties of the rewrite system: parse-time alias resolution, name-based field access, context-free rewrites, and metadata exclusion from computation. A different system with context-sensitive rewrites, position-dependent field access, or metadata that flows into computation would require its own congruence analysis. We do not claim this result generalizes to arbitrary rewrite systems.

4.2 Interpretation

The congruence property answers a practical question: *is it safe to canonicalize a subexpression without considering its pipeline context?*

Without congruence, the answer is no. A local rewrite that is semantically valid in isolation might interact with surrounding pipeline operators to produce a different result. The canonicalization pass would need global analysis of every pipeline context in which a subexpression might appear.

With congruence, the answer is yes. Locally equivalent subexpressions remain equivalent in any well-typed context. The canonicalization pass can operate on subexpressions independently, and the global pipeline identity is preserved.

This is what elevates canonicalization from *local rewrite normalization* to *compositional execution semantics*. Local rewrites compose into global identity guarantees. The 278-to-235 surface-form collapse reported in Paper 2 is safe precisely because the underlying equivalence is a congruence.

Example 4.3 (Congruence in practice)**.** Paper 2 reports that all four surface forms of "rolling z-score of log returns"—including both threading and nesting syntax, both `wzs` and `rol_zsc` names—canonicalize to the same form. The congruence property guarantees that embedding any of these four forms inside a larger pipeline (e.g., as the signal in a risk-adjusted portfolio construction) preserves the equivalence. The larger pipeline's hash is invariant to which surface form the generator happened to produce for the subexpression.

Example 4.4 (Contrast: non-congruent equivalence)**.** Floating-point optimization provides a useful contrast. The compiler transformation $a + b + c \to (a + c) + b$ may be valid in isolation (both produce the same mathematical result) but not under composition: floating-point addition is not associative, so embedding the transformation in a larger numerical pipeline may change the final result. This is a non-congruent equivalence. The system avoids analogous problems because its rewrite rules are syntactic (name-for-name alias substitution, threading elimination) rather than semantic (numerical identity).

5 Quotient Execution Semantics

The congruence established in §4 enables a quotient construction: we can form a category whose objects are equivalence classes rather than individual artifacts. This quotient category provides the formal model of deterministic execution identity.

5.1 Quotient Projection

Definition 5.1 (Quotient projection)**.** The *quotient projection* is the map

$$q_R : \mathcal{B}_R \to \mathcal{B}_R/{\sim_R}$$

defined by $q_R(a) = [a]_R$, the equivalence class of a under $\sim_R$.

Proposition 5.2 (Quotient category)**.** *Given the congruence of $\sim_R$ (Proposition 4.1), the quotient $\mathcal{B}_R/{\sim_R}$ is a well-defined category:*

- Objects*: equivalence classes $[a]_R = \{a' \in \mathrm{Ob}(\mathcal{B}_R) \mid a' \sim_R a\}$.*
- Morphisms*: equivalence classes $[f]_R$ of morphisms.*
- Composition*: $[g]_R \circ [f]_R = [g \circ f]_R$.*

Composition is well-defined: if $f \sim_R f'$ and $g \sim_R g'$, then $g \circ f \sim_R g' \circ f'$ by congruence, so $[g \circ f]_R = [g' \circ f']_R$.

The quotient category provides the paper's formal model of deterministic execution identity. Two specifications are "the same execution" if and only if they belong to the same equivalence class in $\mathcal{B}_R/\sim_R$.

q_R is a functor by construction: it maps objects to their equivalence classes and morphisms to their equivalence classes, and preserves composition and identities.

5.2 Canonical Representative Selection

The quotient projection q_R maps objects to abstract equivalence classes. The implementation needs concrete representatives.

Definition 5.3 (Canonical representative selection)**.** The *canonicalization map* $\kappa_R : \mathrm{Expr}_R \to \mathrm{Expr}_R$ is the implementation's representative-selection operator. For each equivalence class $[a]_R$, it computes a distinguished element:

$$\kappa_R(a) \in [a]_R \tag{1}$$

$$a \sim_R a' \implies \kappa_R(a) = \kappa_R(a') \tag{2}$$

That is: equivalent inputs produce identical (not merely equivalent) canonical representatives.

The distinction between q_R and κ_R matters. q_R maps to equivalence classes (abstract objects in the quotient category). κ_R selects concrete representatives within those classes. q_R is always a functor. κ_R is idempotent ($\kappa_R(\kappa_R(a)) = \kappa_R(a)$) and respects equivalence, but whether it extends to a full functor on morphisms depends on whether it commutes with all morphism compositions.

In the BLISP implementation, the normalize-then-canonicalize pipeline is applied to expressions before pipeline composition, so this commutativity holds for the expression sublanguage. The paper does not assert functoriality of κ_R beyond this scope.

5.3 Hash as Computable Operational Witness

Definition 5.4 (Hash witness)**.** The *content-addressed hash* $H : \mathrm{Expr}_R \to \mathcal{H}$ is a deterministic map from canonical expressions to a hash space (SHA-256 in the implementation).

Proposition 5.5 (Hash witness soundness)**.** *Under the operational assumption of collision-resistant hashing:*

1. Forward direction (exact). $\kappa_R(a) = \kappa_R(a') \implies H(\kappa_R(a)) = H(\kappa_R(a'))$. *Identical canonical forms hash identically, by determinism of* H*.*
2. Reverse direction (operational). $H(\kappa_R(a)) = H(\kappa_R(a'))$ *is treated as operational evidence that* $[a]_R = [a']_R$*. This depends on the collision-resistance assumption: a hash collision would produce a false identity. Under collision resistance, this does not occur in practice.*

Remark 5.6 (Identity vs. witness). The semantic identity is the equivalence class $[a]_R$. The hash $H(\kappa_R(a))$ is its computable operational witness. The quotient category $\mathcal{B}_R/\sim_R$ is defined by $\sim_R$, not by H. Hashing makes the abstract quotient structure operationally observable. The system operationally conflates the two (checking identity means checking hash equality), but they are conceptually distinct.

8-layer structure. Paper 2 decomposes the execution hash into eight semantic layers (registry, specification, morphisms, search, composition, score, selection, data). Each layer corresponds to a quotient at a different level of the execution pipeline. When two end-to-end hashes differ, the per-layer decomposition localizes the divergence to a specific quotient level without re-execution. This layered structure is a direct consequence of the pipeline's compositional architecture: each stage produces a typed output with its own content-addressed identity.

6 Projection Structure and Prompt Fibers

The execution category and its quotient describe the deterministic side of the system. This section connects the stochastic side—prompts, LLM outputs, surface-form specifications—to the deterministic quotient.

6.1 Stochastic Proposal Space

Definition 6.1 (Proposal space)**.** The *stochastic proposal space* $\mathcal{E}_R$ is the set of stochastic realizations—natural-language prompts, LLM outputs, surface-form specifications—that reference capabilities in R.

$\mathcal{E}_R$ is intentionally left weakly structured. It is a set, not a category: we do not impose composition, associativity, or identity on the stochastic upstream. The paper's semantic claims concern deterministic executable identity after projection into the quotient execution category $\mathcal{B}_R/\sim_R$. The asymmetry is by design: the deterministic side is categorical; the stochastic side is a space of inputs.

The elements of $\mathcal{E}_R$ are the inputs to the system. The objects of $\mathcal{B}_R/\sim_R$ are the outputs (execution identities). The core question is: how are they connected?

6.2 Execution Projection

Definition 6.2 (Execution projection)**.** The *execution projection* is a map

$$\pi : \mathcal{E}_R \to \mathcal{B}_R/\sim_R$$

sending each stochastic proposal to its canonical execution class. Concretely, π is the composition of the system's processing pipeline: parse the proposal, validate admissibility, extract the typed specification, normalize, canonicalize, and take the quotient class. π is a set-theoretic map from the proposal space to the quotient category's objects.

Definition 6.3 (Fiber)**.** The *fiber* over an execution class $[b]_R$ is:

$$\pi^{-1}([b]_R) = \{p \in \mathcal{E}_R \mid \pi(p) = [b]_R\}$$

the set of all stochastic proposals that collapse to the same deterministic execution identity.

Fibers measure the multiplicity of the stochastic-to-deterministic mapping. A large fiber means many surface-form proposals collapse to the same execution—the system successfully absorbs generator diversity. A singleton fiber means the proposal space offers no variation for that execution class.

Remark 6.4 (Projection, not fibration)*.* We use the term *projection with fibers* rather than *fibration* in the technical categorical sense. A fibration requires categorical lifting properties that the proposal space does not support—$\mathcal{E}_R$ is a set, not a category. The fiber structure—the decomposition of $\mathcal{E}_R$ into preimages of execution classes—is the operationally relevant property, and it holds by construction of π.

6.3 Collapse Ratio as Fiber Cardinality

Proposition 6.5 (Fiber cardinality and collapse)**.** *The collapse ratio $\gamma = \delta_S/\delta_C$ from Paper 2 [2] measures the ratio of surface-form diversity to the number of distinct execution classes reached. For a set of proposals $P \subseteq \mathcal{E}_R$:*

$$\gamma = \frac{|P|}{|\pi(P)|}$$

where $\pi(P) = \{\pi(p) \mid p \in P\}$ is the image in the quotient. When $\gamma \approx 1$, fibers are approximately singletons: surface diversity maps nearly 1-to-1 onto execution classes.

Paper 2 measured γ across 1,200 LLM generations at four temperature settings. At temperature 0.0, all 30 prompts produce identical specifications across all repetitions ($\gamma = 1$)—fibers are exact singletons. At temperature 0.7, 83% of prompts still collapse to a single canonical specification.

The fibers that are *not* singletons at higher temperatures correspond to genuine intent divergence: the generator proposed different families or metrics, not different surface forms of the same specification. Canonicalization correctly does not collapse these, because they represent distinct execution classes. The collapse ratio cleanly separates surface-form variation (absorbed by canonicalization, within fibers) from intent variation (preserved as distinct execution classes, across fibers).

6.4 Grounding Gate as Fiber Restriction

Proposition 6.6 (Fiber restriction)**.** *The grounding gate* Γ *from Paper 1 [1] restricts the projection to admissible proposals. Define the* admissible proposal subset*:*

$$\mathcal{E}_R^\Gamma = \{p \in \mathcal{E}_R \mid \Gamma(p) = \text{admit}\}$$

The restricted projection $\pi_\Gamma = \pi|_{\mathcal{E}_R^\Gamma}$ *maps only admitted proposals to execution classes. The image* $\pi(\mathcal{E}_R^\Gamma)$ *is a proper subset of* $\text{Ob}(\mathcal{B}_R/{\sim_R})$*: only execution classes reachable from admitted proposals are in the image.*

Paper 1 reported selectivity of 27 admitted out of 96 tuples tested. In the projection framework, this means the grounding gate restricts the reachable execution classes to 27 of 96—those with discovery evidence from the user's terms. The remaining 69 execution classes exist in $\mathcal{B}_R/{\sim_R}$ but are unreachable through the gated projection.

The valid-but-unwarranted failures identified in Paper 1 (failure mode F3) correspond to proposals that land in the wrong fiber. The grounding gate cannot distinguish correct from incorrect selection *within* the admitted set—it restricts which fibers are reachable, not which element within a reachable fiber the generator selects.

7 Deterministic Executable Identity

What does it mean for two executions to be "the same"?

This paper provides a specific, bounded answer. Two objects a and a' in $\mathcal{B}_R$ have the same *deterministic executable identity* if and only if $[a]_R = [a']_R$: they belong to the same equivalence class in the quotient category.

7.1 What Identity Is

Execution identity is membership in the same equivalence class under the declared operational invariances (E1)–(E4). Concretely:

- Two specifications are the same execution if they canonicalize to the same canonical form.
- Two pipeline expressions are the same execution if the system's rewrite rules can transform one into the other.
- Identity is witnessed operationally by hash equality: $H(\kappa_R(a)) = H(\kappa_R(a'))$.
- Identity is compositional: if subexpressions are equivalent, the composed pipelines are equivalent (Proposition 4.1).

7.2 What Identity Is Not

- **Not denotational equivalence.** Two programs may compute the same mathematical function on all inputs but have different canonical forms (and therefore different execution identities) if the rewrite rules do not derive their equivalence. Execution identity is more conservative than semantic equivalence.
- **Not behavioral equivalence.** Two programs may produce the same output on observed inputs by coincidence. Execution identity requires structural derivation through the declared rules, not empirical output comparison.
- **Not universal.** Execution identity is relative to the registry R. Different registries may assign different identities to the same expression, because different alias tables induce different equivalences.

7.3 Description/Identity Separation

Generator (E4) excludes descriptive metadata from execution identity. This creates a clean architectural separation:

Layer	Determines identity	Example
SEM (semantic)	Yes	Operation type, arity
ALG (algebraic)	Yes	Computation rule, decomposition
IMPL (implementation)	Yes	Kernel selection, IR variant
DISC (discovery)	No	Aliases, tags, descriptions

Paper 2 verified this empirically: enriching the descriptions of five pipeline operations produced identical hashes at all eight provenance layers. The categorical interpretation is direct: description changes do not alter equivalence classes because (E4) declares them invariant. The discovery metadata in DISC enables the natural-language interface to evolve—adding aliases, improving descriptions—without invalidating the execution identity of any prior pipeline.

8 Regulated Execution Interpretation

This section assembles the full picture. Each stage of the operational pipeline has a categorical interpretation, and the composition of these stages is the regulated execution path from stochastic proposal to deterministic provenance.

8.1 Pipeline as Categorical Composition

The full system decomposes into a sequence of maps, each with a specific guarantee:

Stage	Map	Signature	Guarantee
Generation	G	$\text{Prompt} \to \mathcal{E}_R$	Produces stochastic proposals
Admission	Γ	$\mathcal{E}_R \to \mathcal{E}_R^\Gamma$	Restricts to admitted proposals
Projection	π	$\mathcal{E}_R^\Gamma \to \text{Ob}(\mathcal{B}_R/{\sim_R})$	Maps proposals to execution classes
Selection	κ_R	$[a]_R \to \kappa_R(a)$	Selects canonical representative
Evaluation	ε_R	$\text{Plan}_R \times \text{Data} \to \text{Result}_R$	Deterministic execution
Witness	H	$\text{Result}_R \to \mathcal{H}$	Content-addressed provenance

The composition $H \circ \varepsilon_R \circ \kappa_R \circ \pi \circ \Gamma \circ G$ is the end-to-end system. The first two stages (G, Γ) are upstream of the canonical execution boundary—G is stochastic, Γ is deterministic but restricts the stochastic input. From π onward, everything is deterministic: the projection maps to equivalence classes, κ_R selects representatives, ε_R evaluates, and H witnesses.

The canonical execution boundary. In Paper 2's terminology, the canonical execution boundary is the point at which the pipeline transitions from stochastic to deterministic. In the categorical framework, this boundary is the projection π: the map from $\mathcal{E}_R$ (the weakly structured proposal space) to $\mathcal{B}_R/{\sim_R}$ (the quotient execution category, where everything is deterministic by construction). Stochasticity does not propagate beyond π.

8.2 Registry Evolution

Proposition 8.1 (Registry monotonicity)**.** *If $R \subseteq R'$ (registry extension: new operations, aliases, or families added), then $\sim_R \subseteq \sim_{R'}$, and there is a natural map*

$$\mathcal{B}_R/{\sim_R} \to \mathcal{B}_{R'}/{\sim_{R'}}$$

sending $[a]_R$ *to* $[a]_{R'}$.

Registry extension may merge equivalence classes (a new alias equates previously distinct expressions) but never splits them.

This formalizes the asymmetry observed in Paper 2's drift detection experiment: adding an alias can change the DIC_HSH (merging classes), but removing an alias can only shrink the equivalence and potentially split classes. The content-addressed hash detects both directions: any change to the registry changes DIC_HSH, alerting the system to re-evaluate execution identity.

Remark 8.2. Registry extension is monotone in equivalence (more aliases means more expressions are equivalent) but not monotone in identity (merged classes change the quotient structure). This is the formal basis for Paper 2's drift detection: a hash change after registry modification is detected at the DIC_HSH layer, precisely because the quotient category changed.

8.3 Provenance as Layered Quotient

The 8-layer execution hash from Paper 2 decomposes into a sequence of quotients, one per semantic layer:

Layer	Hash	Quotient interpretation
1	DIC_HSH	Registry identity: $[R]$
2	FPR_HSH	Specification identity: $[s]_R$
3	MOR_HSHS	Plan identity: $\{[m]_R\}$
4	SRH_HSHS	Search configuration identity
5	CMPN_HSHS	Composition identity
6	SCR_HSH	Score computation identity
7	SEL_HSH	Selection criteria identity
8	DATA_HSH	Data fingerprint

Each layer is a quotient of the corresponding typed objects under the restriction of $\sim_R$ to that type. When two end-to-end hashes differ, comparing the per-layer hashes identifies which quotient changed—fault localization without re-execution. This is a direct consequence of the pipeline's compositional structure: each stage produces typed output with its own content-addressed identity, and the congruence property ensures that per-layer equivalence composes into end-to-end equivalence.

9 Relationship to Papers 1–2

Papers 1–2 empirically establish the operational system. This paper formalizes the semantic structure those systems imply. Every empirical result from the prior papers has a categorical interpretation, and every categorical construction maps to an operational mechanism.

9.1 Mapping Table

9.2 What the Formalization Adds

The empirical systems of Papers 1–2 work without the formalization. Canonicalization produces identical hashes; the grounding gate rejects unwarranted proposals; replay is deterministic. The formalization adds three things:

1. **Compositional justification.** The congruence property (Proposition 4.1) explains *why* local canonicalization is safe in global pipelines. Without this property, the system's correctness would be an empirical observation that might break under pipeline extension. With it, the safety of local rewrites is a structural guarantee.

Table 1: Categorical interpretation of empirical results from Papers 1–2.

Empirical result	Categorical interpretation	Paper
Grounding gate selectivity (27/96 admitted)	Image cardinality of $\pi \circ \Gamma$: 27 reachable execution classes out of 96	1
Valid-but-unwarranted (F3) failure mode	Proposal in wrong fiber: $\pi(p) \neq [b_{\text{intended}}]_R$ within admitted set	1
Gate overhead (<14 ms)	Cost of computing Γ: fiber restriction is cheap relative to evaluation	1
278 surface forms → 235 canonical operations	$\lvert\text{Ob}(\mathcal{B}_R)\rvert = 278$, $\lvert\text{Ob}(\mathcal{B}_R/\!\sim_R)\rvert = 235$: quotient collapses 43 classes	2
Replay determinism (50 runs, all identical)	Fiber singletons: $\lvert\pi^{-1}([b]_R)\rvert = 1$ under deterministic replay	2
Collapse ratio $\gamma \approx 1$ at T=0	Degenerate fibers: generator produces one element per fiber at zero temperature	2
Collapse $\gamma < 1$ at T=0.7	Non-singleton fibers from intent divergence, not surface-form divergence	2
DIC_HSH stability across repetitions	Registry invariant: R unchanged $\implies$ $\mathcal{B}_R/\!\sim_R$ unchanged	2
DIC_HSH change under registry extension	Quotient change under $R \to R'$: merged equivalence classes (Proposition 8.1)	2
Description/identity separation (enrichment → same hashes)	Generator (E4): DISC changes are within equivalence classes, not across them	2
8-layer execution hash	Layered quotient: per-layer content address witnesses per-type equivalence class	2

2. **Precise identity semantics.** The quotient category gives a precise answer to "what does it mean for two executions to be the same?"—a question that Papers 1–2 answer operationally (same hash) but not semantically (same equivalence class under declared invariances).

3. **Architectural vocabulary.** The projection, fibers, and fiber restriction give precise language to concepts that Papers 1–2 describe informally: "collapse," "selectivity," "reachable execution." This vocabulary enables reasoning about system extensions (new families, richer equivalences, multi-registry composition) without re-deriving the operational properties from scratch.

10 Limitations and Non-Claims

This section explicitly bounds what the paper claims.

Not universal semantic equivalence. $\sim_R$ is not behavioral equivalence. Two programs that compute the same function on all inputs are not necessarily equivalent under $\sim_R$ unless the declared rewrite rules derive that equivalence. $\sim_R$ captures the invariances the system explicitly guarantees, not all invariances that happen to hold. This is conservative by design: soundness (equivalent things produce the same hash) is prioritized over completeness (all same things are detected as equivalent).

Not denotational semantics. We do not define program meaning. We define operational identity: when the system treats two artifacts as the same, as determined by its declared transformation rules. The

relationship between operational identity and denotational semantics is a well-studied question in programming language theory; this paper does not address it.

Not rewrite-system completeness. We do not prove that $\sim_R$ captures all equivalences a user might expect. We do not establish confluence (that rewrite order does not matter) or termination (that rewriting always reaches a normal form) as general properties of the rewrite system. The BLISP implementation achieves these in practice through a fixed-order normalization pipeline, but we do not prove them abstractly.

Not a general theorem. The congruence result (Proposition 4.1) depends on specific structural properties of the rewrite system: parse-time alias resolution, name-based field access, context-free rewrites, and metadata exclusion from computation. A different system with context-sensitive rewrites, position-dependent field access, or metadata that flows into computation would require its own analysis. We do not claim congruence holds for arbitrary rewrite systems.

Not geometric. The fibers in §6 are discrete sets, not topological spaces, manifolds, or geometric objects. We do not introduce metric structure, curvature, or continuous mathematics. The fiber language is used in its algebraic sense: a decomposition of a set into preimages.

Not cognitive. The stochastic generator is treated as a black box. We do not model how it produces proposals, why it produces diverse surface forms, or what "understanding" or "intent" means. The formalization starts at the system boundary where proposals arrive, not inside the generator.

Category theory is analytical, not foundational. The operational system works without the formalization. Category theory provides analytical vocabulary—it reveals structure and enables compositional reasoning. It does not change the system's behavior or extend its capabilities. The system was designed, built, and empirically validated (Papers 1–2) before this formalization was written.

Single domain. The execution category is defined in terms of the BLISP system operating in systematic trading research. The categorical constructions are domain-independent (categories, quotients, and projections do not depend on the application domain), but the congruence analysis depends on the specific rewrite rules, and those rules are BLISP-specific. Demonstrating congruence in other domains (molecular simulation, climate modeling, etc.) would require analyzing each domain's rewrite system independently.

Registry-relative means contingent. Every equivalence class, every execution identity, and every fiber is relative to a specific registry R. Changing R changes the quotient category. This is not a weakness of the formalism; it is a faithful representation of the operational reality. The system's identity judgments are contingent on its current state.

11 Conclusion

Papers 1–2 built the operational system: a grounding gate that enforces admissibility, and a canonical execution pipeline that provides deterministic replay identity. This paper reveals the semantic structure those systems imply.

The central result is that the operational equivalence generated by the system's canonicalization rules is a congruence—equivalent subexpressions remain equivalent in any well-typed pipeline context. This property is what makes local canonicalization safe in global pipelines. Without it, every local rewrite would require global analysis. With it, the 278-to-235 surface-form collapse and the 50-run replay determinism reported in Paper 2 are not just empirical observations but consequences of a compositional guarantee.

The quotient category $\mathcal{B}_R/\sim_R$ gives precise meaning to "deterministic execution identity." The projection from stochastic proposals to the quotient, with its fiber structure, gives precise meaning to "collapse" and "selectivity." The grounding gate is a fiber restriction; the 8-layer hash is a layered quotient; registry evolution

is a quotient morphism. Each operational mechanism has a categorical interpretation, and each categorical construction maps to an empirical measurement.

The semantics are deliberately constrained. They formalize what the system declares, not what programs mean. They are operational, registry-relative, and execution-grounded. They do not claim universal equivalence, denotational completeness, or geometric structure. The category theory is analytical: it reveals the compositional structure that the empirical system exhibits, and it provides vocabulary for reasoning about extensions—richer invariance algebras, multi-registry composition, dynamic registry evolution—without re-deriving the operational properties from scratch. Paper 4 [3] extends this analysis to compositional provenance semantics over the quotient category.

References

[1] T. Dionysopoulos. The grounding gate: Admissibility and replay guarantees for AI-driven research. *Zenodo*, 2026. `doi:10.5281/zenodo.20456984`.

[2] T. Dionysopoulos. Canonical execution semantics for stochastic program generators. *Zenodo*, 2026. `doi:10.5281/zenodo.20457255`.

[3] T. Dionysopoulos. Provenance algebra for deterministic AI execution. *BLISP Research Program*, Paper 4, 2026.

[4] E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In *LISA*, 2004.

[5] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[6] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[7] S. Awodey. *Category Theory*. Oxford University Press, 2nd edition, 2010.

[8] M. Zaharia et al. Accelerating the machine learning lifecycle with MLflow. *IEEE Data Engineering Bulletin*, 41(4), 2018.

[9] H. Chase. LangChain, 2023.

[10] S. Yao et al. ReAct: Synergizing reasoning and acting in language models. In *ICLR*, 2023.

[11] Q. Wu et al. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv:2308.08155*, 2023.

[12] B. Carpenter et al. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1), 2017.