The Semantic Structure of Execution: An Empirical Study of Predictive Coordinates in Computational Operations

Thomas Dionysopoulos, CFA

Abstract

Computational operations in execution systems are typically classified by what they compute (sum, average, sort) rather than by their mathematical relationship to data. Type systems constrain which operations are valid; they do not predict how an optimizer will treat them. We ask whether a low-dimensional classification of operations by their data-access pattern—which we call a *semantic coordinate*—can function as a predictive structural coordinate of an execution space.

We introduce DependencyClass, a 7-class partition of operations by data-access pattern: element-local, lagged, windowed, cumulative, recursive, global, and masking. We report an empirical study on a corpus of 61 operations in Blisp, a deterministic execution system for temporal computation. DependencyClass functions as a predictive semantic coordinate for four independent optimizer behaviors—elementwise fusibility, fusion boundary placement, warmup classification, and mask-aware dispatch—with 99.6% accuracy (243/244 predictions; one documented exception). On a held-out set of 25 operations not used in any prior analysis, prediction accuracy is 100% (100/100). Against 1,000 random 8-class partitions, DependencyClass is a 13-sigma outlier ($z = 13.0$, $p < 10^{-38}$). An information-theoretic analysis shows that DependencyClass explains 96.7–100% of optimizer behavior entropy. A coordinate ablation confirms that DependencyClass is the dominant predictive coordinate among those studied: removing access to it reduces accuracy from 99.6% to below the majority baseline. The coordinate is partially recoverable from behavior alone (NMI = 0.933), and it encodes more structure than the four measured behaviors require.

The contribution is not a naming convention. It is the empirical demonstration that a single low-dimensional coordinate, derived from the mathematical data-access pattern of an operation, contains nearly all information an optimizer needs to determine that operation's treatment. The causal structure is: operation → semantic coordinate → behavior. Blisp is the experimental vehicle that exposes this coordinate at the language surface; the finding concerns the coordinate structure itself.

1 Introduction

When a computational system encounters a new operation, the optimizer must decide how to treat it. Can it be fused with adjacent operations into a single pass? Does it require a warmup period before producing valid output? Must it operate on a masked subset of the data? These decisions are currently made per-operation: each new operation requires explicit optimizer annotations, manual classification, or pattern-matching rules that grow linearly with the operation vocabulary.

This paper asks a different question: *can a single structural property of the operation predict all of these decisions simultaneously?*

The property we investigate is the operation's *data-access pattern*—the mathematical relationship between each output element and the input elements it depends on. An elementwise operation reads one input to produce one output. A windowed operation reads a contiguous range. A cumulative operation reads all preceding elements. These patterns are mathematical properties of the operation, independent of any particular implementation or optimizer.

We formalize this property as DependencyClass, a 7-class partition:

We call this a *semantic coordinate*: a low-dimensional variable derived from an operation's mathematical structure that locates the operation within a space of execution behaviors. The term distinguishes our object of study from three superficially related concepts. A *type* constrains what inputs an operation accepts; a

Class	Data-access pattern
Elementwise	Each output depends on exactly one input at the same position
Pointwise	Each output depends on one input at a fixed offset
Windowed	Each output depends on a contiguous range of inputs
Prefix	Each output depends on all preceding inputs
Recursive	Each output depends on the previous output
Global	Output depends on the entire input (or multiple series)
Mask	Sets metadata rather than computing values

coordinate predicts how the optimizer treats it. A *symbol* is a representation; a coordinate is the underlying classification the symbol may or may not encode. An *annotation* is declared by the programmer; a coordinate is derivable from the operation's mathematical definition.

1.1 Invariants and structural coordinates

Many mathematical disciplines organize complex objects by identifying low-dimensional invariants that predict behavior: genus in topology, dimension in linear algebra, degree in algebraic geometry, order in group theory. The invariant compresses the space of objects into a small set of classes while preserving information about the properties that matter for a given purpose.

This paper investigates whether computational operations admit a similar structural description. We do not claim that DependencyClass is an invariant of computation in the formal mathematical sense—it is not preserved under arbitrary transformations, and it is defined on a specific operation space rather than a general category. We investigate it as a *candidate structural coordinate*: a low-dimensional classification, derived from mathematical properties, that empirically predicts optimizer behavior with high accuracy. The question is whether this execution space has enough structure that a single coordinate can do most of the work.

1.2 Summary of findings

The central finding is that DependencyClass—a single coordinate with 7 values—functions as a predictive semantic coordinate for four independent optimizer behaviors, achieving 99.6% accuracy across 61 operations. The result is supported by seven experiments:

1. **Holdout prediction.** Rules frozen on 36 operations predict all four behaviors correctly on 25 held-out operations (100/100).

2. **Random baseline.** Among 1,000 random 8-class partitions, none matches DependencyClass. The z-score is 13.0 ($p < 10^{-38}$).

3. **Information-theoretic analysis.** DependencyClass explains 96.7–100% of optimizer behavior entropy, measured by mutual information. Even after conditioning on correlated behaviors, DependencyClass explains 95–100% of remaining entropy (Section 7.4).

4. **Coordinate ablation.** DependencyClass alone achieves 99.6% accuracy. The next-best single coordinate (ObservationModel) achieves 84.0%. Combining coordinates does not improve over DependencyClass alone.

5. **Representation comparison.** Three encodings of the same 36 operations—one that exposes semantic coordinates, one that uses Python source code, and one that uses opaque identifiers—achieve 100%, 75%, and 0% accuracy respectively.

6. **Coordinate recovery.** Clustering operations by observed behavior yields 6 clusters that align with DependencyClass at NMI = 0.933. Three of 8 classes are uniquely recoverable; two class-pairs merge because they share identical behavior on the four measured predicates.

7. **Foundation analysis.** A systematic enumeration of 15 candidate coordinates shows 5 are fully redundant with DependencyClass, 2 are constant, and only DependencyClass achieves perfect prediction across the four core optimizer behaviors measured.

1.3 Contributions

The paper makes three contributions:

- **Empirical:** A single 7-class coordinate functions as a predictive semantic coordinate for four optimizer behaviors, generalizes to unseen operations, and is a strong outlier over random classifications.
- **Information-theoretic:** DependencyClass contains 96.7–100% of the mutual information between operation identity and optimizer behavior for the four measured predicates.
- **Methodological:** A protocol for evaluating whether an operation classification is predictive (holdout, random baseline, information content, ablation, recovery) rather than merely descriptive.

1.4 Scope and limitations

The study is conducted on a single system (Blisp) with a single optimizer. The 61 operations are domain-specific (temporal finance). The four measured behaviors are optimizer-specific. We do not claim that DependencyClass generalizes to other systems—we claim that the methodology demonstrates genuine predictive content in this system, and that the data-access pattern classification is mathematically grounded rather than system-specific. (See Appendix D for a detailed threats-to-validity analysis.)

1.5 Outline

Section 2 positions this work within the Blisp research program. Section 3 defines semantic coordinates formally. Section 4 states the predictive hypothesis. Section 5 describes the experimental design. Sections 6 to 10 report the results. Section 11 states precisely what was and was not measured. Section 12 covers related work. Section 13 discusses implications. Section 14 concludes.

2 Background and Motivation

2.1 The Blisp Research Program

This paper is the sixth in a research program studying the structure of AI-generated computation. The preceding papers established:

Paper 1 (The Grounding Gate) introduced a mandatory admissibility boundary between AI-proposed operations and deterministic execution. A live capability registry (236 capabilities) validates that every proposed operation has evidence in a discovery result. Valid-but-unwarranted executions are reduced from 23.3% to 10.0% (Fisher exact $p = 0.027$).

Paper 2 (Canonical Execution Semantics) showed that stochastic program generators produce surface-form variation that masks execution equivalence. A canonicalization pipeline collapses 278 surface forms to 235 canonical operations.

Paper 3 (Execution Categories) formalized execution equivalence as a congruence on the operation space and constructed the quotient category.

Paper 4 (Provenance Algebra) decomposed execution provenance into dependency-indexed layers.

Paper 5 (Execution Fibers) measured the fiber structure of the map from stochastic proposals to deterministic execution.

2.2 The Gap

Papers 1–5 study the map from stochastic proposals to deterministic execution: how proposals are validated (Paper 1), how surface forms collapse to canonical forms (Paper 2), when two operations are equivalent (Paper 3), what provenance each execution carries (Paper 4), and how many proposals map to each execution (Paper 5).

None of these papers asks: *what structure exists within the deterministic execution space itself?* After canonicalization collapses surface variation, 235 distinct operations remain. Are these 235 operations an unstructured set, or do they organize into classes with predictable properties?

This paper provides empirical evidence for the latter. A single coordinate—DependencyClass—partitions operations into 7 classes such that class membership determines optimizer behavior with near-perfect accuracy. Papers 1–5 study the vertical structure (proposals to execution). Paper 6 studies the horizontal structure (within execution).

2.3 Prior Approaches

Existing approaches to operation classification—type systems, compiler IRs, abstract interpretation, semantic annotations, polyhedral analysis, and effect systems—are discussed in detail in Section 12. The key distinction is that these approaches either constrain what the optimizer *may* do (types, effects) or extract exact dependencies from code structure (polyhedral analysis). Semantic coordinates instead classify operations by the *kind* of data-access pattern they exhibit and ask whether this classification predicts optimizer behavior empirically.

3 Semantic Coordinates

3.1 Definition

Let Σ be a set of computational operations. A *semantic coordinate* is a function $C : \Sigma \to V$, where V is a finite set, such that:

1. **Derivability.** $C(\sigma)$ is determinable from the mathematical definition of σ, without reference to any particular implementation or optimizer.
2. **Compression.** $|V| \ll |\Sigma|$. The coordinate compresses the operation space.
3. **Predictiveness.** There exists a behavior function $B : \Sigma \to W$ and a prediction function $f : V \to W$ such that $B(\sigma) = f(C(\sigma))$ for most or all $\sigma \in \Sigma$.

Condition 1 distinguishes coordinates from annotations (which are declared, not derived). Condition 2 distinguishes coordinates from identity (where $C(\sigma) = \sigma$ is trivially predictive). Condition 3 distinguishes coordinates from arbitrary classifications.

A coordinate satisfying all three conditions is *predictive*. A classification satisfying only 1 and 2 is *descriptive*—it compresses but does not predict. A classification satisfying only 2 and 3 is *engineered*—it predicts but may be an artifact of the optimizer rather than a mathematical property of the operation.

The causal structure under investigation is:

$$\text{Operation} \longrightarrow \text{Semantic Coordinate} \longrightarrow \text{Optimizer Behavior}$$

The coordinate is derived from the operation's mathematical properties. The behavior is predicted from the coordinate. Any symbolic representation (name, annotation, database entry) that encodes the coordinate is a *carrier* of the predictive variable, not the variable itself.

Class	Mathematical characterization	Example
Elem	$\text{out}[i] = g(\text{in}[i])$	abs, log, exp
Pointwise	$\text{out}[i] = g(\text{in}[i], \text{in}[i-k])$ for fixed k	dlog, shift
Windowed	$\text{out}[i] = g(\text{in}[i-w..i])$ for fixed w	rolling_avg
Prefix	$\text{out}[i] = g(\text{in}[0..i])$	cumsum, scan
Recursive	$\text{out}[i] = g(\text{in}[i], \text{out}[i-1])$	ema, locf
Global	$\text{out} = g(\text{in}_1, \ldots, \text{in}_n)$	align, covariance
Mask	Sets `active_mask` metadata	weekend_mask
Selection	Selects a subset of rows	keep

3.2 DependencyClass

DependencyClass assigns each operation to one of 8 classes based on its data-access pattern:

The 8 classes are ordered by inclusion of the dependency set:

$$\text{Elem} \subset \text{Pointwise} \subset \text{Windowed} \subset \text{Prefix} \subset \text{Global}$$

with Recursive forming a separate branch (each output depends on the previous output, creating a sequential dependency chain). Mask and Selection are structural rather than computational: they modify metadata or filter rows rather than computing values.

3.3 Additional Coordinates

Three additional coordinates were investigated:

Linearity classifies whether the operation preserves linear combinations of inputs: Linear (averaging, summation) vs. Nonlinear (logarithm, standard deviation). This coordinate is partially independent of DependencyClass: 4 of 7 classes contain both linear and nonlinear operations. The DepClass × Linearity matrix has 79% occupancy (11 of 14 cells).

ShapeEffect classifies the operation's effect on the output schema: Same (row count preserved), ReduceScalar, ChangeSchema, ReduceRows, AddMask. Within the 36-operation NumericFunc space, ShapeEffect is constant (all Same), so it provides zero information.

ObservationModel classifies how the operation references past values: NoRef, PrevOne, PrevK, RollingWindow, AllPrevious, AllData. This coordinate closely tracks DependencyClass—each ObservationModel value maps to exactly one DepClass. It is redundant for the four measured behaviors.

3.4 Symbolic Representation

In Blisp, semantic coordinates are encoded in operation symbols at the IR level. The symbol `SHF_WIN_LIN_AVG` encodes:

- `SHF`: shift-equivariant (constant across all operations)
- `WIN`: DependencyClass = Windowed
- `LIN`: Linearity = Linear
- `AVG`: Operation identity = rolling average

This encoding makes coordinates accessible to pattern matching without code analysis. However, the symbolic encoding is a *representation choice*, not the predictive variable.

One encoding inconsistency was discovered during the study: the operation SPR (pairwise spreads) carries the symbol `SHF_PTW_LIN_SPR`, suggesting DependencyClass = Pointwise. Its mathematical properties are Global (cross-column dependency, no row lag). Assigning DependencyClass from the mathematical definition (Global) produces correct predictions; assigning from the symbol (Pointwise) produces two errors. This confirms that the predictive variable is the coordinate, not the symbol that encodes it.

4 Predictive Hypothesis

Hypothesis 1. *DependencyClass functions as a predictive semantic coordinate for optimizer behavior. Given an operation's DependencyClass, it is possible to predict: (1) whether the operation is elementwise-*

fusible, (2) whether it creates a fusion boundary, (3) what warmup class it requires, and (4) whether it requires mask-aware dispatch.

4.1 Prediction Rules

The following rules were frozen before the holdout experiment:

DependencyClass	Elementwise	Fusion breaks	Warmup	Mask-aware
Elem	true	false	None	false
Pointwise	false	true	Lag	true
Windowed	false	true	Window	true
Prefix	false	true	Full	true
Recursive	false	true	Full	true
Global	false	true	None	false
Mask	false	true	None	false
Selection	false	true	None	false

These rules encode a single principle: the data-access pattern determines whether the optimizer can treat the operation as self-contained (Elementwise) or must account for cross-row dependencies. The specific predictions follow from the data-access pattern:

- *Elementwise*: only element-local operations can fuse.
- *Fusion boundary*: any cross-row dependency breaks fusion.
- *Warmup*: the data-access pattern determines how many initial outputs are undefined (lag = k, window = w, prefix/recursive = full series).
- *Mask-aware dispatch*: operations that read across rows must skip masked positions to avoid corrupting the computation.

4.2 Expected Exceptions

Two classes have edge cases. **Global** operations (align, covariance) do not require warmup despite reading the full input, because they compute a single global result rather than a streaming output. **Mask** and **Selection** operations modify metadata or filter rows; they do not compute values from cross-row data, so they do not need mask-aware dispatch. These exceptions are predicted by the rules and confirmed by the optimizer.

5 Experimental Design

5.1 Corpus

The experimental corpus consists of 61 operations in the Blisp IR:

DependencyClass	Count	Examples
Elem	16	LOG, ABS, ADD, MUL, GTR, SIGN, EXP, SQRT
Pointwise	6	DLOG, RET, SHF, LAG_OBS
Windowed	16	AVG, SDV, MAX, MIN, RSK_ADJ, MAX_IND
Prefix	5	SUM, SCAN, CUM_PROD
Recursive	3	EMA, LOCF, EWM_SDV
Global	12	ALIGN, COV, COR, EIG, ZSCORE
Mask	1	MSK_WKE
Selection	2	KEEP, KEEP_ALIGNED

The corpus was divided into a **training set** (36 operations) used in the foundation analysis and representation comparison, and a **holdout set** (25 operations) used exclusively in the holdout prediction experiment. All prediction rules were developed using only the training set.

The holdout set spans binary arithmetic (8), matrix algebra (6), windowed indicators (2), cross-axis operations (3), ternary (1), schema transformations (1), and hypothetical operations (5). The hypothetical operations do not yet exist in Blisp; their DependencyClass is assigned from mathematical necessity, constituting testable predictions.

5.2 Ground Truth

Ground truth for all four behaviors was extracted from the Blisp optimizer source code: `is_pure_elementwise()` (`src/ir.rs:917-931`), `fusion_breaks()` (`src/capabilities.rs:1227-1234`), `warmup_class` (`src/exec.rs:156-611`), and `mask_aware_dispatch` (`src/exec.rs:171`). For hypothetical operations, ground truth was determined by mathematical necessity.

5.3 Seven Experiments

1. **Foundation analysis.** Enumerated 15 candidate coordinates. Tested redundancy, independence, and clustering. Purpose: identify which coordinates carry information.
2. **Representation comparison.** Encoded 36 operations in three representations (coordinate-exposing, Python source, opaque). Purpose: test whether coordinates—not representation—are predictive.
3. **Holdout prediction.** Froze rules from Experiment 1. Applied to 25 held-out operations. Purpose: test generalization.
4. **Coordinate ablation.** Predicted 4 behaviors using each coordinate alone and in combinations. Purpose: quantify contribution.
5. **Random baseline.** Generated 1,000 random DependencyClass assignments (uniform over 8 classes, deterministic LCG, seed 42). Purpose: test whether any random partition achieves comparable accuracy.
6. **Information-theoretic analysis.** Computed I(DependencyClass; Behavior) for each behavior. Purpose: quantify information content.
7. **Coordinate recovery.** Clustered 61 operations by behavior vector. Computed NMI between clusters and DependencyClass. Purpose: test whether DependencyClass is latent in behavior data.

5.4 Experimental Protocol

All experiments are implemented as Rust tests in the Blisp repository (`tests/semantic_coordinate_*.rs`). They are additive: no existing test, optimizer logic, planner logic, execution code, or public API was modified. Each experiment can be removed with a single commit revert. Prediction rules were frozen after Experiment 1 and before Experiment 3. No rule was modified after observing holdout results.

6 Results: Prediction Accuracy

6.1 Holdout Prediction

The 25 held-out operations were classified by DependencyClass from their mathematical properties. The frozen prediction rules produced 100 predictions (25 operations $\times$ 4 behaviors).

Result: 100/100 correct (100%).

Zero false predictions. Zero coordinate failures. The prediction rules developed on 36 training operations generalize perfectly to 25 unseen operations spanning every DependencyClass.

Category	Operations	Predictions	Correct
Binary arithmetic	8	32	32
Matrix algebra	6	24	24
Windowed indicators	2	8	8
Cross-axis	3	12	12
Ternary	1	4	4
Schema	1	4	4
Hypothetical	5	20	20

Behavior	Correct	Total	Accuracy
is_elementwise	61	61	100.0%
fusion_breaks	61	61	100.0%
warmup_class	61	61	100.0%
mask_aware	60	61	98.4%
Total	**243**	**244**	**99.6%**

6.2 Full Corpus Accuracy

Across all 61 operations and 4 behaviors (244 predictions):

The single exception is KEEP_ALIGNED (Selection class). The prediction rule maps Selection to mask_aware = false, but KEEP_ALIGNED has mask_aware = true. This operation combines row selection with temporal alignment, a hybrid that requires mask awareness despite its Selection classification. The exception is documented, not hidden.

6.3 Representation Comparison

The same 36 training operations were encoded in three representations and tested on 6 optimizer behaviors (216 predictions per representation):

Representation	Correct	Total	Accuracy
Coordinates exposed (BLISP)	216	216	100.0%
Python source code	162	216	75.0%
Opaque identifiers	0	216	0.0%

The Python representation achieves 75% through code-level inference: `x.rolling(w).mean()` implies a window dependency; `np.log(x)` implies elementwise. But Python fundamentally cannot predict mask-aware dispatch (0/36), because mask awareness is a runtime property of the execution system that has no counterpart in Python's computation model.

The opaque representation (`OP001`, `OP002`, ...) achieves exactly 0/216. Without access to any coordinate information, no prediction is possible.

6.4 Interpretation

The 100% to 0% ablation between coordinate-exposing and coordinate-removing representations confirms: the coordinates are the sole carrier of predictive information. The symbol is the carrier; the coordinate is the cause.

7 Information-Theoretic Analysis

7.1 Mutual Information

For each of the 4 optimizer behaviors B, we compute the mutual information $\mathrm{I}(\textsc{DependencyClass}; B) = \mathrm{H}(B) - \mathrm{H}(B \mid \textsc{DependencyClass})$:

Behavior	$\mathrm{H}(B)$	$\mathrm{H}(B \mid \textsc{DependencyClass})$	$\mathrm{I}(\textsc{DependencyClass}; B)$	**% Explained**
is_elementwise	0.830	0.000	0.830	100.0%
fusion_breaks	0.830	0.000	0.830	100.0%
warmup_class	1.716	0.000	1.716	100.0%
mask_aware	1.000	0.033	0.967	96.7%

DependencyClass explains 100% of the entropy for three of four behaviors. For mask_aware, the residual entropy (0.033 bits) is entirely attributable to the keep_aligned exception.

7.2 Linearity's Information Content

Linearity carries less than 3% of the mutual information for all four behaviors. DependencyClass is the dominant coordinate among those studied, by a factor of approximately 40×. (Linearity predicts other decisions not measured here—e.g., which role an operation plays in zscore fusion—but these are not among the four behaviors under study.)

7.3 Joint Information

Adding Linearity to DependencyClass does not increase prediction accuracy (both achieve 243/244). The joint mutual information $\mathrm{I}(\textsc{DependencyClass}, \mathrm{Lin}; B)$ equals $\mathrm{I}(\textsc{DependencyClass}; B)$ for all four behaviors. Linearity is conditionally independent of these behaviors given DependencyClass.

7.4 Conditional Mutual Information

Two of the four behaviors are logically correlated: is_elementwise and fusion_breaks are complementary, and mask_aware partially tracks fusion_breaks. To address whether DependencyClass provides information *beyond* what correlated behaviors already reveal, we compute conditional mutual information.

$\mathrm{I}(\textbf{DependencyClass}; \textbf{warmup} \mid \textbf{is\_ew}) = 1.394$ **bits.** Given is_elementwise = true (16 operations), warmup is always None. Given is_elementwise = false (45 operations), warmup has four possible values. The conditional entropy H(warmup | is_ew) = 1.394 bits. DependencyClass explains 100% of this remaining entropy: $\mathrm{H}(\text{warmup} \mid \textsc{DependencyClass}, \text{is\_ew}) = 0.000$ bits. Knowing that an operation is non-elementwise tells you nothing about *which* kind of non-elementwise it is. DependencyClass resolves this completely.

$\mathrm{I}(\textbf{DependencyClass}; \textbf{mask} \mid \textbf{fb}) = 0.627$ **bits.** Given fusion_breaks = true (45 operations), mask_aware is true for 31 and false for 14. DependencyClass explains 95.0% of this remaining entropy, with the residual (0.033 bits) again attributable to the keep_aligned exception.

Even after removing the information shared with correlated behaviors, DependencyClass retains 95–100% of the remaining predictive content.

8 Ablation Studies

8.1 Single-Coordinate Ablation

Each coordinate was tested as the sole predictor on all 61 operations across 4 behaviors (244 predictions):

Coordinate	Correct	Accuracy
Majority baseline	152/244	62.3%
ShapeEffect only	35/244	14.3%
ObservationModel only	205/244	84.0%
DependencyClass only	**243/244**	**99.6%**

8.2 Additive Ablation

No combination of additional coordinates improves over DependencyClass alone. The keep_aligned exception is not resolved by any coordinate in the system—it requires operation-level knowledge.

8.3 Implication

Among the coordinates studied, DependencyClass is both necessary and sufficient for predicting these four optimizer behaviors. It is necessary because removing it drops accuracy below the majority baseline. It is sufficient because adding other coordinates does not improve accuracy. This dominance is not guaranteed by the coordinate definitions. A priori, Linearity might distinguish elementwise operations that can fuse from those that cannot. The empirical finding is that, for these four behaviors, DependencyClass captures all the variation.

9 Coordinate Recovery

9.1 Behavioral Clustering

Each operation is represented by a 4-dimensional behavior vector: (is_elementwise, fusion_breaks, warmup_class, mask_aware). Operations with identical behavior vectors form a cluster. Six distinct clusters emerge:

Cluster	Behavior signature	DepClasses	Ops
1	ew=T, fb=F, warm=None, mask=F	{Elem}	16
2	ew=F, fb=T, warm=Lag, mask=T	{Pointwise}	6
3	ew=F, fb=T, warm=Win, mask=T	{Windowed}	16
4	ew=F, fb=T, warm=Full, mask=T	{Prefix, Recursive}	8
5	ew=F, fb=T, warm=None, mask=F	{Global, Mask, Sel}	14
6	ew=F, fb=T, warm=None, mask=T	{Selection}	1

9.2 Recovery Quality

NMI(DependencyClass, BehaviorCluster) = 0.933. Elem, Pointwise, and Windowed each map to exactly one cluster (39 of 61 operations, 63.9% uniquely recoverable). Prefix and Recursive merge (same warmup = Full). Global, Mask, and Selection merge (same warmup = None, mask = false).

9.3 Interpretation

The merges are informative. DependencyClass distinguishes 8 classes, but only 6 have distinct optimizer behavior. The two merges show that DependencyClass encodes *more* structure than these four behaviors require. This is evidence against circularity. If DependencyClass were engineered to match the optimizer, it would have exactly 6 classes. The fact that it has 8 suggests that DependencyClass captures mathematical structure that exists independently of the optimizer.

10 Random Baseline

To test whether DependencyClass's predictive power is achievable by chance, we generated 1,000 random DependencyClass assignments using a deterministic LCG (seed 42). Each was scored using the frozen prediction rules.

Metric	Blisp	Random (1,000 trials)
Score	243/244 (99.6%)	Mean: 133.5/244 (54.7%) Std: 8.4 Max: 145/244 (59.4%)
z-score	**13.0** ($p < 10^{-38}$)	

0 of 1,000 random trials matched or exceeded Blisp's score. The gap is largest for warmup_class (100% vs. 33.1%) because warmup has 4 classes while the other behaviors are binary.

11 What Was Measured

To prevent over-interpretation, we state precisely what was and was not measured.

Measured: prediction accuracy of DependencyClass for four categorical optimizer behaviors; holdout generalization to 25 unseen operations; information content (mutual information, conditional entropy, conditional mutual information); recovery of DependencyClass from observed behavior (NMI); statistical separation from 1,000 random partitions (z-score); coordinate ablation; representation comparison.

Not measured: execution cost, runtime, or memory consumption; energy consumption; cross-system generalization; continuous performance properties; pipeline-level behavior; universality across domains.

All claims in this paper are restricted to what was measured. The finding is that DependencyClass functions as a predictive semantic coordinate for four specific categorical behaviors in one system. Generalization beyond this scope is an open question.

12 Related Work

12.1 Operation Classification in Compilers

Compiler IRs classify operations for optimization. LLVM classifies instructions by opcode into categories that determine which optimization passes apply. MLIR introduces *traits*—declarative properties like `Commutative`, `NoSideEffect`, `Elementwise`—that operations declare and optimization passes query. Our work differs in two ways: (a) DependencyClass is a single coordinate that predicts multiple behaviors, rather than a set of independent traits; (b) DependencyClass is derivable from mathematical properties, while MLIR traits are declared annotations.

12.2 The TVM/Relay Classification

The TVM/Relay framework [7] classifies tensor operations by pattern for fusion scheduling. This is the most directly comparable prior work.

Relay TOpPattern	Description	Nearest DepClass
kElemWise	Elementwise operations	Elem
kBroadcast	Broadcast elementwise	Elem (with shape)
kInjective	One-to-one, shape change	Pointwise or Selection
kCommReduce	Commutative reductions	Global or Prefix
kOutEWiseFusable	EW-fusible output	(composite rule)
kTuple	Tuple operations	(structural)

Our contribution beyond Relay is methodological. Relay's classification is stated and used by the optimizer; it is not empirically validated with holdout prediction, random baselines, information-theoretic analysis, or coordinate recovery. Whether Relay's TOpPattern would achieve similar results under this protocol is an open question and a natural direction for future work.

12.3 Type-Directed Optimization

Session types [6], refinement types [4], and liquid types [8] carry semantic information that constrains optimization. These systems constrain what the optimizer *may* do; semantic coordinates predict what the optimizer *will* do. The distinction is between legality (types) and prediction (coordinates).

12.4 Abstract Interpretation and Polyhedral Analysis

Cousot and Cousot [2] introduced abstract interpretation: computing on abstract values that approximate concrete program behavior. DependencyClass could be formalized as an abstract domain over data-access patterns. We do not pursue this formalization, but note that our empirical results (96.7–100% entropy explained) suggest the abstraction is surprisingly precise.

The polyhedral model [3, 1] represents loop dependencies as integer linear constraints. It is more precise than DependencyClass but requires affine loop structure. DependencyClass applies to black-box operations (recursive filters, matrix decompositions) that the polyhedral model cannot analyze.

12.5 Effect Systems and Semantic Annotations

Effect systems [5] track computational effects at the type level. Effects describe what an operation *does to the world*; DependencyClass describes what an operation *reads from its input*.

Semantic annotations (Java `@Override`, Python `@jit`, Halide schedules) attach optimizer-relevant metadata by programmer declaration. Semantic coordinates differ: they are *derivable* from the operation's mathematical definition.

12.6 Structural Invariants in Related Fields

The idea that complex objects can be organized by low-dimensional structural properties has a long history across mathematics and science.[1]

In *topology*, the genus of a surface and the fundamental group of a space are invariants that determine which continuous transformations are possible. In *algebra*, the order and structure of a group determine its representation theory. In *dimensional analysis*, the Buckingham Π theorem shows that physical systems with many variables often depend on a small number of dimensionless groups.

DependencyClass is not an invariant in the formal mathematical sense: it is not preserved under arbitrary operation transformations, and we do not define a group of transformations under which it is stable. The parallel is structural rather than formal. Like a topological invariant, DependencyClass compresses a large space (61 operations) into a small classification (7 classes) while retaining nearly all information relevant to a specific set of questions (optimizer behavior). The information-theoretic result (96.7–100% entropy explained) can be read as an empirical compression bound: the operation space, for the purpose of optimizer behavior, has an effective dimensionality of approximately 1 coordinate.

13 Discussion

13.1 Why DependencyClass Works

DependencyClass classifies operations by their data-access pattern. This pattern constrains optimization because optimizers must respect data dependencies. The prediction rules follow from the mathematical properties of the data-access pattern:

[1]The broader research motivation of the Blisp program is the search for structural invariants beneath the apparent variation in computational systems—invariant structure that persists across surface forms, execution models, and stochastic generation processes.

- Fusion is legal when output computations are independent.
- Warmup reflects the initialization period before valid output.
- Mask-aware dispatch is required when the operation reads across rows.

This structural argument suggests that DependencyClass may function as a predictive coordinate in other optimizers that respect data dependencies. We cannot verify this empirically without testing on additional systems, but the mathematical basis is independent of Blisp.

13.2 What DependencyClass Is Not

DependencyClass is not a type system—it does not constrain which operations are valid. It is not a cost model—it does not predict execution time. It is not a correctness proof—it predicts what the optimizer *will do*, not what it *should do*. It is not unique—other partitions might achieve similar predictive power. The random baseline ($z = 13.0$) shows that random partitions fail dramatically, but we have not exhaustively searched the space of possible partitions.

13.3 Operation, Coordinate, Symbol

Three objects must be distinguished:

- **Operation:** the mathematical computation (e.g., "rolling average over a window of size w"). This is the real object.
- **Coordinate:** the structural classification derived from the operation's mathematical properties (e.g., DependencyClass = Windowed). This is the predictive variable.
- **Symbol:** the representation that may or may not encode the coordinate (e.g., `SHF_WIN_LIN_AVG`). This is the carrier.

The causal chain is:

$$\text{Operation} \rightarrow \text{Coordinate} \rightarrow \text{Behavior}$$

The operation determines the coordinate. The coordinate determines the behavior. The symbol encodes the coordinate but does not cause the behavior. Three experiments establish this: the representation comparison (Section 6), the SPR encoding inconsistency (Section 3), and the information-theoretic analysis (Section 7).

13.4 Structural Coordinates and Invariants

The central empirical result of this paper is not that a label predicts behavior. It is that the operation space admits a low-dimensional structural coordinate—derived from mathematical properties, not optimizer internals—that determines optimizer behavior with near-perfect accuracy.

Three properties distinguish this from feature engineering:

1. **Derivability.** DependencyClass is determined by the mathematical definition of the operation, not by training a model on optimizer outputs.
2. **Recoverability.** DependencyClass is partially recoverable from behavior alone (NMI = 0.933). The coordinate is latent in the structure of the execution space, not imposed on it.
3. **Compression with near-zero information loss.** The 61-operation space compresses to 7 classes with 96.7–100% of behavior entropy retained.

These three properties—derivable, recoverable, compressive—place the finding closer to invariant discovery than to feature engineering. We are cautious about the invariant analogy. DependencyClass is not preserved under arbitrary transformations of the operation space. But the pattern—a low-dimensional classification, derived from intrinsic mathematical properties, that predicts behavior and is partially recoverable from that behavior—resembles invariant discovery more closely than it resembles supervised classification or feature selection.

13.5 Explicit vs. Reconstructed Coordinates

Traditional systems require *semantic reconstruction*: the optimizer must recover optimizer-relevant properties from the operation's implementation. Blisp takes a different approach: semantic coordinates are exposed directly at the language surface. This paper does not claim that explicit coordinates are universally superior to reconstruction. It demonstrates that when explicit coordinates are available, they can carry nearly all information an optimizer needs for categorical decisions.

13.6 Implications for the Blisp Research Program

Paper 6 provides the missing structural component of the research program. Papers 1–5 study the vertical structure (proposals → execution). Paper 6 studies the horizontal structure (within execution). Paper 2 collapses surface variation; Paper 6 characterizes the *residual* variation. Paper 3 defines when two operations are *equivalent*; Paper 6 defines what makes two operations *different*.

14 Conclusion

This paper provides empirical evidence that a computational operation space can admit low-dimensional structural coordinates that predict optimizer behavior.

A single semantic coordinate—DependencyClass, a 7-class partition by data-access pattern—explains 96.7–100% of optimizer behavior entropy for four predicates across 61 operations. It generalizes to 25 unseen operations with 100% accuracy. It is a 13-sigma outlier over 1,000 random classifications. It is partially recoverable from observed behavior alone (NMI = 0.933), and it encodes more mathematical structure than the measured behaviors require. Even after conditioning on correlated behaviors, DependencyClass explains 95–100% of remaining entropy.

The predictive variable is the coordinate, not the symbol that carries it. The coordinate is derived from the operation's mathematical properties—its data-access pattern—not from the optimizer's implementation. The operation determines the coordinate. The coordinate determines the behavior. The symbol is merely the carrier.

The methodology—holdout prediction, random baseline, information-theoretic analysis, conditional mutual information, coordinate ablation, and coordinate recovery—provides a reusable protocol for evaluating whether any operation classification is genuinely predictive rather than merely descriptive.

The significance of this result is not that a particular coordinate was found to be useful in a particular system. It is that the coordinate is derivable from mathematical structure, predicts optimizer behavior, is partially recoverable from that behavior, and compresses the operation space with near-zero information loss. These properties—derivability, predictiveness, recoverability, compression—are characteristic of structural invariants in many mathematical disciplines. We observe that the data-access pattern plays a role in this execution space analogous to the role that genus plays in surface topology or dimension plays in linear algebra: a low-dimensional quantity, derived from intrinsic properties, that determines much of the object's behavior.

The study has clear limitations: a single optimizer, a single domain, co-evolution between the coordinate system and the optimizer, and correlated behaviors (Appendix D). These scope the claim. Within that scope, the evidence supports a specific conclusion: the execution space studied here has low-dimensional predictive structure. Whether other execution spaces admit similar structure remains open; the methodology provides a protocol for finding out. When such structure exists, identifying it is a problem of invariant discovery, not feature engineering.

A Experimental Data

A.1 Full Operation Corpus (Training Set, 36 Operations)

A.2 Random Baseline Distribution

Summary statistics for 1,000 random trials: mean = 133.5/244, std = 8.4, min = 107/244, median = 135/244, max = 145/244. Blisp score: 243/244. z-score: 13.0.

Operation	DepClass	Lin	is_ew	breaks	warmup	mask
LOG	Elem	NLN	T	F	None	F
ABS	Elem	NLN	T	F	None	F
EXP	Elem	NLN	T	F	None	F
SQRT	Elem	NLN	T	F	None	F
SIGN	Elem	NLN	T	F	None	F
INV	Elem	NLN	T	F	None	F
CLAMP	Elem	NLN	T	F	None	F
TANH_SAT	Elem	NLN	T	F	None	F
DLOG_OBS	PTW	NLN	F	T	Lag	T
DLOG_OFS	PTW	NLN	F	T	Lag	T
RET	PTW	NLN	F	T	Lag	T
SHF	PTW	LIN	F	T	Lag	T
SHF_FWD	PTW	LIN	F	T	Lag	T
LAG_OBS	PTW	LIN	F	T	Lag	T
AVG	WIN	LIN	F	T	Win	T
AVG_EXCL	WIN	LIN	F	T	Win	T
MIN2_AVG	WIN	LIN	F	T	Win	T
MIN2_AVG_EXCL	WIN	LIN	F	T	Win	T
SDV	WIN	NLN	F	T	Win	T
SDV_EXCL	WIN	NLN	F	T	Win	T
MAX	WIN	NLN	F	T	Win	T
MIN	WIN	NLN	F	T	Win	T
RSK_ADJ	WIN	NLN	F	T	Win	T
MAX_IND	WIN	NLN	F	T	Win	T
MIN_IND	WIN	NLN	F	T	Win	T
ZSC	WIN	NLN	F	T	Win	T
SUM	PFX	LIN	F	T	Full	T
SUM0	PFX	LIN	F	T	Full	T
SCAN	PFX	NLN	F	T	Full	T
EXP_AVG	REC	LIN	F	T	Full	T
LOCF	REC	NLN	F	T	Full	T
MSK_WKE	MSK	NLN	F	T	None	F
KEEP	SEL	NLN	F	T	None	F
KEEP_ALIGNED	SEL	NLN	F	T	None	**T***
ALIGN	GBL	LIN	F	T	None	F
ASOF_ALIGN	GBL	LIN	F	T	None	F

*KEEP_ALIGNED is the single exception: Selection → mask_aware = false mispredicts the actual behavior (mask_aware = true).

A.3 Mutual Information Calculations

Entropy computations use base-2 logarithms (bits).

is_elementwise: 16 true, 45 false. $H(B) = 0.830$ bits. $H(B|\text{DependencyClass}) = 0.000$.

fusion_breaks: 45 true, 16 false. $H(B) = 0.830$ bits. $H(B|\text{DependencyClass}) = 0.000$.

warmup_class: None=29, Lag=6, Window=16, Full=10. $H(B) = 1.716$ bits. $H(B|\text{DependencyClass}) = 0.000$.

mask_aware: true=34, false=27. $H(B) = 1.000$ bits. $H(B|\text{DependencyClass}) = 0.033$ bits (from Selection class having 1 true and 1 false).

A.4 Conditional Mutual Information

$I(\text{DependencyClass}; \text{warmup} \mid \text{is\_ew})$: $H(\text{warmup} \mid \text{ew=T}) = 0$; $H(\text{warmup} \mid \text{ew=F}) = 1.889$ bits; $H(\text{warmup} \mid \text{is\_ew}) = 1.394$ bits; $H(\text{warmup} \mid \text{DependencyClass}, \text{is\_ew}) = 0.000$ bits. Result: 1.394 bits. **100% of remaining entropy.**

$I(\text{DependencyClass}; \text{mask} \mid \text{fb})$: $H(\text{mask} \mid \text{fb=F}) = 0$; $H(\text{mask} \mid \text{fb=T}) = 0.894$ bits; $H(\text{mask} \mid \text{fb}) = 0.660$ bits; $H(\text{mask} \mid \text{DependencyClass}, \text{fb}) = 0.033$ bits. Result: 0.627 bits. **95.0% of remaining entropy.**

A.5 Coordinate Recovery — NMI Calculation

NMI = $\mathrm{I}(\textsc{DependencyClass}; BC)/\sqrt{\mathrm{H}(\textsc{DependencyClass}) \cdot \mathrm{H}(BC)}$, where $\mathrm{H}(\textsc{DependencyClass})$ = 2.665 bits, $\mathrm{H}(BC)$ = 2.320 bits, $\mathrm{I}(\textsc{DependencyClass}; BC)$ = 2.320 bits. NMI = 2.320/2.487 = 0.933.

B Coordinate System Details

B.1 Redundancy

Five candidate coordinates are fully redundant with DependencyClass: WindowRequirement ≡ (DepClass = Windowed), RequiresLag ≡ (DepClass = Pointwise), Statefulness ≡ (DepClass ∈ {Prefix, Recursive}), ParameterDependency ≡ (DepClass ∈ {Pointwise, Windowed}), Equivariance = constant (all shift-equivariant, no variation).

C Relationship to Papers 1–5

Paper	Question	Paper 6 connection
1. Grounding Gate	Are proposed ops legitimate?	Coordinate structure is predictive
2. Canonical Exec.	When are two proposals the same?	Characterizes residual variation
3. Exec. Categories	What is the equivalence structure?	Complementary partition structure
4. Provenance Alg.	How does provenance decompose?	DependencyClass indexes provenance layers
5. Exec. Fibers	How many proposals per exec?	Structure within execution space

Papers 1–5: vertical structure (proposals → execution). Paper 6: horizontal structure (within execution).

D Threats to Validity

D.1 Single Optimizer (Construct Validity)

All four behaviors are properties of Blisp's optimizer. We have not tested whether DependencyClass functions as a predictive coordinate in other systems (pandas, polars, DuckDB, Halide). The mathematical argument—that data-access patterns constrain any correct optimizer—is plausible but not empirically demonstrated. This is the study's most significant limitation.

D.2 Single Domain (External Validity)

The 61 operations are all temporal finance operations. While they span diverse mathematical types, they share a common structure (time-indexed series). The coordinate system may not extend to operations in other domains without modification.

D.3 Co-evolution (Internal Validity)

DependencyClass and the Blisp optimizer were built by the same team. The coordinate system and the optimizer co-evolved. Circularity cannot be fully excluded. Three pieces of evidence mitigate this concern: (a) holdout prediction succeeds on operations not considered during coordinate design; (b) the coordinate recovery experiment shows DependencyClass has more resolution than the optimizer uses (8 classes vs. 6 behavior clusters); (c) the warmup prediction is emergent—warmup logic was implemented for correctness, not designed around DependencyClass.

Full elimination of co-evolution would require an independent researcher to assign DependencyClass to operations in a system they did not build.

D.4 Correlated Behaviors (Statistical Validity)

The four behaviors are not fully independent. is_elementwise and fusion_breaks are logically related. Counting 244 predictions overstates the effective degrees of freedom. The conditional mutual information analysis (Section 7.4) addresses this directly: even after conditioning on correlated behaviors, DependencyClass explains 95–100% of remaining entropy.

D.5 Coordinate Selection Bias (Internal Validity)

DependencyClass was selected from 15 candidate coordinates after empirical evaluation. The mitigation is that DependencyClass was not selected from a large search space of possible partitions—it was one of 15 pre-defined candidates based on known mathematical properties.

D.6 Small Corpus (Statistical Power)

61 operations is the full Blisp IR vocabulary. The $z = 13.0$ result is statistically robust ($p < 10^{-38}$), but a larger corpus would strengthen the result. The sample IS the population for this system, which limits the concern about sampling error but does not address generalizability.

D.7 No Alternative Partitions (Conclusion Validity)

The random baseline proves DependencyClass $\gg$ random, not that DependencyClass is optimal. The recovery experiment (NMI = 0.933) suggests that any high-accuracy partition must closely resemble DependencyClass, but this is not formally proven.

References

[1] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, 2008.

[2] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[3] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[4] T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, 1991.

[5] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *LFP*, 1986.

[6] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, 1998.

[7] J. Roesch, S. Lyubomirsky, L. Weber, J. Pollock, M. Kirisame, T. Chen, and Z. Tatlock. Relay: A new IR for machine learning frameworks. In *MAPL*, 2019.

[8] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.