Semantic Coordinates as Predictive Objects in Time-Series Computation

Thomas Dionysopoulos, CFA

Abstract

Optimizing compilers and query planners must decide, for each operation they encounter, which transformations are legal: whether to fuse, what warmup to require, whether mask state must be tracked, and where to place optimization boundaries. These decisions are typically implemented as per-pass checks against per-operation properties. We ask an empirical question that, to our knowledge, prior work has not addressed: how much of an optimizer's behavior is determined by a single categorical property of the operations it processes?

We study this question in Blisp, a time-series intermediate representation where each operation carries an explicit semantic coordinate encoding its data-dependency shape. We classify 61 operations into seven categories by their dependency structure: elementwise, pointwise, windowed, prefix, recursive, global, and mask. We then measure how well this single coordinate—called DependencyClass—predicts four independent optimizer behaviors: fusion eligibility, fusion boundary placement, warmup classification, and mask awareness.

DependencyClass predicts 243 of 244 behavior instances (99.6% accuracy, $z = 13.0$ versus random baseline, $p < 10^{-38}$), removing 96.7–100% of behavioral entropy as measured by mutual information. On a 25-operation holdout set with frozen prediction rules, accuracy is 100/100. The single misprediction (Keep_Al, a selection operation predicted as not mask-aware but actually mask-aware) is explained by a second coordinate—observation stream—which resolves it without modifying the primary coordinate.

Four independently-developed subsystems in Blisp—the fusion optimizer, the pipeline validator, the capability reasoner, and the execution dispatcher—all dispatch on dependency shape despite having been implemented separately. We position this finding as the first empirical measurement of how much optimizer behavior is determined by operation classification, complementing the definitional approach taken by systems such as MLIR traits and TVM's `OpPattern`.

This paper does not claim that the presented coordinate system is unique, minimal, or universal. It claims only that explicit semantic coordinates can be predictive of execution behavior, and that this relationship can be measured.

1 Introduction

When an optimizer encounters an operation, it must answer several questions. Can this operation be fused with its neighbors? Does it require a warmup period before producing valid output? Does it interact with masked observations? Will it break an ongoing fusion chain? These questions determine the quality of the generated execution plan.

In most systems, the answers are encoded implicitly. The fusion pass contains a match statement that lists fusable operations. The validator contains a separate match statement for warmup rules. The executor contains yet another for mask-aware dispatch. Each subsystem maintains its own understanding of each operation's behavior, and consistency across subsystems is maintained by engineering discipline rather than by structure.

We investigate an alternative: encoding the answers in the operation's identity itself. If an operation's name contains a structured semantic coordinate—a small categorical property describing its computational character—then the questions above may be answerable by inspecting the coordinate rather than consulting operation-specific logic.

This idea is not new in isolation. MLIR attaches traits such as `Commutative` and `Elementwise` to operations, and passes query these traits to make optimization decisions [1]. TVM's Relay framework assigns each operation an `OpPattern` from a six-valued enum that drives its fusion algorithm [2]. Futhark

classifies operations as second-order array combinators (SOACs) and uses the classification to determine fusion legality [3]. In each case, a semantic property on an operation gates a specific optimization pass.

What has not been measured, to our knowledge, is the *predictive power* of such properties. In existing systems, the relationship between property and behavior is definitional: MLIR's `Elementwise` trait does not *predict* that an operation will be fusable; it *defines* fusability for the passes that check it. Each trait predicts one pass behavior, by construction. The question we ask is different: can a *single* semantic property predict *multiple independent* behaviors, and can this predictive relationship be quantified?

We present evidence from Blisp, a time-series intermediate representation in which operation identities are structured as serialized semantic coordinate tuples. The variant name `SHF_WIN_LIN_AVG` encodes three coordinates: `SHF` (shift-equivariant), `WIN` (windowed dependency shape), `LIN` (linear). We focus on one coordinate—dependency shape, which we call DependencyClass—and measure how well it predicts four optimizer behaviors across 61 operations.

1.1 Contributions

We make three claims, each supported by completed experiments:

Claim 1. A single 7-valued coordinate (DependencyClass) predicts four independent optimizer behaviors with 99.6% accuracy (243/244) and removes 96.7–100% of behavioral entropy as measured by mutual information.

Claim 2. The predictive relationship generalizes to unseen operations: 100/100 on a 25-operation holdout set with frozen prediction rules, including 5 hypothetical operations not yet implemented.

Claim 3. This is, to our knowledge, the first empirical measurement of how much optimizer behavior is determined by operation classification in a compiler or query-processing setting.

1.2 Scope

This paper studies semantic coordinates as predictive objects. It does not claim that DependencyClass is the unique, minimal, or universal coordinate for time-series operations. It does not claim that the coordinate system generalizes beyond the domain studied. It does not claim superiority over existing systems. The contribution is the measured relationship between a semantic coordinate and execution behavior, and the demonstration that this relationship can be quantified using standard information-theoretic tools.

2 Background

2.1 Time-Series Operations

Time-series computation involves a diverse set of operations on ordered, indexed data. These range from elementwise transforms (logarithm, sign, absolute value) through cross-row operations (differences, rolling averages, cumulative sums, exponential moving averages) to whole-dataset operations (covariance matrices, cross-sectional rankings).

Each class of operation interacts differently with optimizers. An elementwise operation can be fused with adjacent elementwise operations into a single pass over the data. A rolling window operation cannot be fused in this way because it must maintain a sliding window of w previous values. A cumulative sum cannot be fused because each output depends on all preceding inputs. These differences are well understood by practitioners but are rarely formalized as properties of the operations themselves.

2.2 Optimizer Behaviors

We study four behavioral dimensions that an optimizer must determine for each operation it encounters:

Fusion eligibility (`is_elementwise`). Whether an operation can participate in operator fusion—the merging of consecutive operations into a single kernel to eliminate intermediate allocations. Only operations that process each element independently, without cross-row dependencies, are fusion-eligible.

Fusion boundary (`fusion_breaks`). Whether an operation terminates an ongoing fusion chain. Operations that are not elementwise break fusion boundaries, preventing the optimizer from fusing operations across them.

Warmup class (`warmup_class`). The number of input rows required before an operation produces its first valid output. Elementwise operations require zero warmup. Pointwise operations (e.g., first differences) require a fixed lag k. Windowed operations require their full window size w. Prefix and recursive operations (cumulative sums, exponential moving averages) require the entire preceding history.

Mask awareness (`mask_aware`). Whether an operation must respect an active observation mask. In time-series computation, certain rows may be marked as ineligible (e.g., weekend rows in financial data). Mask-aware operations skip masked rows in their cross-row computations; non-mask-aware operations process all rows uniformly.

These four behaviors are logically independent. Knowing that an operation is not elementwise does not determine its warmup class (it could be lag, window, or full). Knowing that an operation breaks fusion does not determine whether it is mask-aware. Any predictive relationship among them is an empirical finding, not a logical necessity.

2.3 The Blisp Intermediate Representation

Blisp is a time-series computation system in which expressions are parsed, normalized, canonicalized, planned into an intermediate representation (IR), optimized, and executed. The IR represents computations as directed acyclic graphs (DAGs) of typed operations.

Each operation in the IR is identified by a variant of the `NumericFunc` enum, which encodes semantic coordinates in its name. For example:

Variant Name	Dependency	Linearity	Operation
`SHF_WIN_LIN_AVG`	Windowed	Linear	Rolling average
`SHF_PTW_OBS_NLN_DLOG`	Pointwise	Nonlinear	Log-difference
`SHF_PFX_LIN_SUM`	Prefix	Linear	Cumulative sum
`SHF_REC_EXP_AVG`	Recursive	Nonlinear	EMA
`MSK_WKE`	Mask	—	Weekend mask
`LOG`	Elementwise	Nonlinear	Natural logarithm

Table 1: Example IR variant names with encoded semantic coordinates.

The naming convention `SHF_{DEP}_{OBS}_{LIN}_{NAME}` is not documentation attached to an otherwise opaque identifier. It *is* the identifier. The Rust type system requires exhaustive matching over these variants, meaning that every subsystem that processes the IR must account for every coordinate combination that exists.

2.4 Prior Art: Semantic Properties in Optimizers

Several systems attach semantic properties to operations and use them in optimization:

MLIR [1] defines operation traits such as `Pure` (no side effects), `Commutative`, `Elementwise`, `IsIdempotent`, and `IsInvolution`. Passes query traits via `op->hasTrait<T>()`. The CSE pass requires `Pure`; the linalg fusion pass requires all-parallel iterator types; the DCE pass requires `NoMemoryEffect`. Each trait gates one or a few specific passes. The relationship is definitional: the trait IS the decision.

TVM/Relay [2] assigns each operation an `OpPatternKind` from a six-valued enum: `kElemWise`, `kBroadcast`, `kInjective`, `kReduction`, `kOutEWiseFusable`, `kOpaque`. The fusion algorithm computes `CombinePattern` = $\max(\text{lhs}, \text{rhs})$ and applies ceiling rules. `OpPattern` is a single coordinate that drives one behavior (fusion).

Futhark [3] classifies operations as Second-Order Array Combinators (SOACs): `map`, `reduce`, `scan`, `scatter`, `hist`. Fusion rules form an explicit matrix over SOAC pairs: map-map fuses, map-reduce fuses, scan-scan does not. The SOAC classification determines fusion legality.

Mathematica [4] attaches attributes such as `Orderless` (commutative), `Flat` (associative), and `Listable` (auto-maps over lists) to symbols. The pattern matcher and evaluator consume these attributes mechanically.

In each system, semantic properties are **definitional**: the property defines the behavior for the subsystem that checks it. No system, to our knowledge, has empirically measured how much of an optimizer's behavior is determined by its operation properties, nor demonstrated that a single property predicts multiple independent subsystems simultaneously.

We classify prior art by the role that semantic properties play:

Level	Description	Examples
1	Labels exist	All systems
2	Labels are first-class	MLIR, TVM, Futhark, Mathematica
3	Labels drive execution	MLIR traits, TVM OpPattern, Futhark SOACs
4	Labels predict behavior (one-to-many)	—
5	Predictive power empirically measured	—

Table 2: Taxonomy of semantic property roles. The present work operates at Level 5.

3 Method

3.1 Corpus Construction

We construct a corpus of 61 operations from Blisp's IR. Each operation is a distinct variant of the `NumericFunc`, `BinaryFunc`, `JoinOp`, `SchemaOp`, `MatrixOp`, or `AggOp` enum. For each operation, we record:

The coordinate. DependencyClass, a categorical variable with 8 values: Elementwise (Elem), Pointwise (Ptw), Windowed (Win), Prefix (Pfx), Recursive (Rec), Global (Gbl), Mask (Msk), and Selection (Sel). Assignment is based on the qualitative structure of the operation's data-dependency function: how many input positions each output position depends on, and in what pattern.

DependencyClass	Data Dependency $d(f)(i)$	Count
Elementwise	$\{i\}$ — current position only	16
Pointwise	$\{i{-}k\}$ — fixed offset	6
Windowed	$\{i{-}w{+}1, \ldots, i\}$ — bounded window	14
Prefix	$\{0, \ldots, i\}$ — unbounded prefix	2
Recursive	$\{0, \ldots, i\}$ with internal state	2
Global	all positions	11
Mask	metadata modification	2
Selection	column/row selection	8

Table 3: DependencyClass categories with data-dependency patterns and corpus counts.

Ground truth. Four binary or categorical behaviors, extracted from the Blisp codebase by inspecting the relevant subsystem:

- **is_elementwise**: whether the operation appears in the `is_pure_elementwise()` predicate of the fusion optimizer (`ir_fusion.rs`).
- **fusion_breaks**: whether the operation breaks a fusion chain, i.e., is NOT elementwise.
- **warmup_class**: the number of input rows consumed before first valid output, classified as None (0), Lag (k), Window (w), or Full (all preceding).
- **mask_aware**: whether the operation's execution kernel reads the `active_mask` field and operates on the eligible observation stream rather than all rows.

Ground truth is determined by code inspection, not by running the system. Each behavior is traced to a specific code location: fusion eligibility to `ir_fusion.rs:99`, mask awareness to dispatch arms in `exec.rs`, warmup class to kernel structure, and fusion boundary to the complement of the elementwise predicate.

3.2 Prediction Rules

Prediction rules are fixed functions from DependencyClass to each behavior. They were written once and frozen before any accuracy measurement:

```
predict_elementwise(dc)   = (dc == Elem)
predict_fusion_breaks(dc) = (dc != Elem)
predict_warmup(dc) = match dc {
    Elem | Global | Mask | Selection => None,
    Pointwise                        => Lag,
    Windowed                         => Window,
    Prefix | Recursive               => Full,
}
predict_mask_aware(dc) = dc in {Ptw, Win, Pfx, Rec}
```

These rules encode a simple hypothesis: the data-dependency pattern of an operation determines its optimizer behavior. Elementwise operations (no cross-row dependency) are fusable and not mask-aware. Operations that reference previous rows (Pointwise through Recursive) are mask-aware because they must respect the eligible observation stream. Warmup is determined by the span of the dependency: lag k for Pointwise, window w for Windowed, and all preceding rows for Prefix and Recursive.

3.3 Evaluation Metrics

Prediction accuracy. The number of correct predictions out of 244 (61 operations $\times$ 4 behaviors).

Random baseline. We generate 1,000 random permutations of the DependencyClass assignments across the 61 operations and measure prediction accuracy for each. The z-score compares the actual accuracy to this null distribution.

Mutual information. For each behavior B, we compute the mutual information $\mathrm{I}(\mathrm{DC}; B) = \mathrm{H}(B) - \mathrm{H}(B|\mathrm{DC})$, where H is Shannon entropy. The percentage $\mathrm{H}(B)$ explained is $\mathrm{I}(\mathrm{DC}; B)/\mathrm{H}(B)$.

Conditional mutual information. We measure whether the four behaviors are redundant or carry independent information. Specifically, we compute $\mathrm{I}(B_i; \mathrm{DC} \mid B_j)$—how much DependencyClass still tells you about behavior i after you already know behavior j.

Coordinate ablation. We measure prediction accuracy using subsets of a four-coordinate system: DependencyClass, Linearity, ShapeEffect, and ObservationModel. This tests whether DependencyClass is privileged or whether any coordinate would do.

Representation comparison. We compare three representations of operations for behavior prediction: (a) semantic coordinates (the DependencyClass value), (b) Python source code inspection (reading the kernel implementation), and (c) opaque identifiers (operation names with no structural information).

3.4 Holdout Protocol

We construct a holdout set of 25 operations that were not used during rule development. The prediction rules are frozen—no rule is modified after seeing holdout results. The holdout includes:

- 8 binary arithmetic/comparison operations (Elem)
- 6 matrix operations (Global)
- 2 windowed index operations (Windowed)
- 2 cross-axis operations (Global)
- 1 ternary operation (Global)
- 1 schema operation (Global)
- 5 hypothetical operations not yet implemented (Win, Pfx, Rec)

The hypothetical operations (rolling median, rolling skewness, running product, running maximum, exponentially weighted moving standard deviation) have ground-truth behaviors determined by mathematical necessity rather than code inspection. For example, a running product must be prefix-dependent, not fusable, mask-aware, and have full warmup—regardless of implementation.

4 Results

4.1 Main Result

DependencyClass predicts 243 of 244 behavior instances correctly, for an overall accuracy of 99.6%.

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%**

Table 4: Per-behavior prediction accuracy. Three behaviors are predicted perfectly; the single error is in mask_aware (Keep_Al).

Three behaviors are predicted perfectly. The single error occurs in mask_aware: Keep_Al (a selection operation that preserves alignment) is predicted as not mask-aware but is actually mask-aware in the implementation. We analyze this error in section 5.1.

4.2 Random Baseline

Under 1,000 random permutations of DependencyClass assignments, the mean accuracy is 133.5/244 (54.7%) with standard deviation 8.4. The maximum observed is 145/244. No random assignment achieved 243/244.

The z-score for the actual result is:

$$z = \frac{243 - 133.5}{8.4} = 13.0$$

The corresponding p-value is less than 10^{-38}. The probability of achieving 243/244 by chance is effectively zero.

4.3 Information-Theoretic Analysis

Behavior	H(B)	H(B|DC)	I(DC; 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%

Table 5: Mutual information analysis. DependencyClass removes all uncertainty about three behaviors and 96.7% of uncertainty about mask_aware.

DependencyClass removes all uncertainty about is_elementwise, fusion_breaks, and warmup_class. It removes 96.7% of uncertainty about mask_aware, with the residual 0.033 bits attributable to the Keep_Al exception.

4.4 Conditional Independence

The four behaviors might appear to carry redundant information (for instance, is_elementwise and fusion_breaks are complementary by definition). To test whether DependencyClass captures independent structure, we measure conditional mutual information:

Conditioning	Target	CMI (bits)	% of target H
is_elementwise known	warmup_class	1.394	100.0%
fusion_breaks known	mask_aware	0.627	95.0%

Table 6: Conditional mutual information: DependencyClass resolves behavioral uncertainty beyond what other behaviors already reveal.

Knowing is_elementwise already tells you that an operation does or does not require warmup—but it does not tell you *which* warmup class applies (Lag vs. Window vs. Full). DependencyClass resolves this entirely (1.394 bits, 100% of remaining entropy). Similarly, knowing fusion_breaks does not determine mask_aware; DependencyClass resolves 95% of what remains. The four behaviors are not redundant with respect to DependencyClass.

4.5 Coordinate Ablation

Coordinate Set	Correct	Total	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%
DC + Linearity	243	244	99.6%
DC + ShapeEffect	243	244	99.6%
DC + ObservationModel	243	244	99.6%
All four coordinates	243	244	99.6%

Table 7: Coordinate ablation. DependencyClass alone achieves maximum accuracy; adding other coordinates does not improve it.

DependencyClass alone achieves the maximum accuracy. Adding any other coordinate does not improve it. ObservationModel alone reaches 84.0%, confirming that it captures partial structure but is dominated by DependencyClass. The majority baseline (always predicting the most common class) reaches only 62.3%.

4.6 Holdout Generalization

On the 25-operation holdout set, with prediction rules frozen from the main experiment:

Category	Operations	Predictions	Correct
Binary arithmetic	4	16	16
Binary comparison	4	16	16
Matrix	6	24	24
Windowed	2	8	8
Cross-axis	2	8	8
Ternary	1	4	4
Schema	1	4	4
Hypothetical	5	20	20
Total	**25**	**100**	**100**

Table 8: Holdout generalization. All 100 predictions correct, including 20 predictions for 5 hypothetical (unimplemented) operations.

All 100 predictions are correct, including the 20 predictions for 5 hypothetical operations that have not been implemented.

4.7 Representation Comparison

We compare three representations for behavior prediction on a subset of 54 operations (the intersection of all three evaluation methods):

Representation	Correct	Total	Accuracy
Semantic coordinates (DependencyClass)	216	216	100.0%
Python source code	162	216	75.0%
Opaque identifiers	0	216	0.0%

Table 9: Representation comparison. Coordinates achieve perfect prediction; source code achieves 75%; opaque names achieve 0%.

Semantic coordinates achieve perfect prediction. Inspecting the Python source code (kernel implementations) achieves 75%—much better than chance but far from the coordinate-based prediction. Opaque identifiers provide zero behavioral information.

5 Analysis

5.1 The Keep_Al Misprediction

The single error in the main experiment is Keep_Al, a selection operation that preserves temporal alignment. DependencyClass assigns it to Selection (elementwise dependency shape—output[i] depends only on input[i]). The prediction rule for Selection yields `mask_aware` = false. But Keep_Al is implemented as mask-aware: it reads the eligible observation stream.

The explanation lies in a second coordinate that DependencyClass does not capture: **observation stream**. Keep_Al has elementwise dependency shape but interacts with the eligible observation stream (EOS) rather than operating on all rows uniformly. The dependency shape is correct (each output position depends only on the corresponding input position), but the *stream on which positions are defined* differs from the default.

This misprediction reveals a refinement opportunity: a two-coordinate system (DependencyClass, ObservationStream) would resolve the error without modifying the primary coordinate. We note that this is exactly how scientific models are refined—a misprediction reveals a missing variable, and the model is extended rather than discarded. We leave this extension to future work.

5.2 Why One Coordinate Suffices for Four Behaviors

The conditional mutual information analysis (section 4.4) provides the explanation. The four behaviors are not independent: they share latent structure. DependencyClass captures this shared structure.

Consider the prediction rules. An elementwise operation is fusable (no cross-row dependency to break fusion), has no warmup (each output is immediately computable), and is not mask-aware (it processes each row independently of the observation stream). All four predictions follow from a single property: the absence of cross-row dependency. Similarly, a windowed operation is not fusable (it maintains a sliding window), has warmup proportional to the window size, and is mask-aware (it must skip masked rows in its window). All four predictions follow from the presence and structure of cross-row dependency.

DependencyClass captures nearly all the information relevant to each operation's interaction with the optimizer. The four behaviors are different projections of the same underlying property.

5.3 Operational Consumption of Coordinates

The semantic coordinate encoded in each operation's variant name is not a passive label. It is actively consumed by four independently-developed subsystems in Blisp:

Fusion optimizer. The function `is_pure_elementwise()` in `ir_fusion.rs` determines fusion eligibility by matching on operation variants. Only operations with no dependency-shape prefix (bare names like `LOG`, `ABS`, `EXP`) match—precisely the Elementwise category.

Pipeline validator. The function `check_plan_warnings()` in `validate.rs` fires six diagnostic warnings based on dependency-shape predicates: `is_recursive_node()` matches `SHF_REC_*` variants, `is_rolling_node()` matches `SHF_WIN_*` variants, `is_mask_node()` matches `MSK_*` variants, `is_locf_node()` matches `SHF_REC_NLN_LOCF`. These predicates pattern-match on the dependency coordinate embedded in the variant name.

Capability reasoner. The function `derive_semantics()` in `capabilities.rs` determines mask handling by matching on coordinate substrings: `contains("SHF_WIN")`, `contains("SHF_PFX")`, `contains("SHF_PTW")`, `contains("SHF_REC")`, `contains("MSK")`. The coordinate substring in the variant name IS the semantic dispatch key.

Execution dispatcher. The match block in `exec.rs` routes operations to kernels based on their variant name: `SHF_WIN_*` variants route to `apply_rolling_mask_aware`, `SHF_PTW_*` to `apply_shift_oriented` or `apply_dlog_oriented`, `SHF_PFX_*` to `apply_cumsum_oriented`, `SHF_REC_*` to `apply_ema` or `apply_locf`. The dependency coordinate determines the kernel.

These four subsystems were written at different times for different purposes. The fusion optimizer was written to eliminate intermediate allocations. The validator was written to catch common pipeline errors. The capability reasoner was written to expose operation semantics to external agents. The executor was written to dispatch operations to kernels. That all four converge on the same coordinate—dependency shape—is the empirical finding. It is not guaranteed by construction: the validator could have dispatched on linearity, the capability reasoner on output shape, the executor on arity. They did not.

5.4 From Labels to Predictive Objects

We distinguish three roles that a semantic property can play:

Descriptive label. A human-readable annotation attached after implementation. Example: a docstring that says "this is a rolling operation." The label can be changed without affecting behavior.

Definitional gate. A property that is part of the decision rule for a specific subsystem. Example: MLIR's `Elementwise` trait determines fusion eligibility. Removing the trait changes behavior, but only for the subsystem that checks it.

Predictive object. A property that determines behavior across multiple independently-developed subsystems. The property was not designed for any single subsystem; it captures shared structure that all subsystems independently exploit. Removing the property degrades multiple subsystems simultaneously.

DependencyClass operates as a predictive object in Blisp. It was not designed for the fusion optimizer specifically—it captures the data-dependency structure that all four subsystems need to know. The evidence: (a) a single coordinate predicts four behaviors, (b) alternative coordinates fail (ObservationModel alone achieves 84.0%), (c) the four subsystems were developed independently, and (d) the prediction generalizes to unseen operations.

6 Discussion

6.1 Relationship to Prior Art

The distinction between Blisp and prior systems is not that Blisp has semantic properties (MLIR, TVM, Futhark all do). The distinction is structural:

MLIR uses approximately 10 binary traits, each gating one or a few optimization passes. `Pure` gates CSE. `Elementwise` gates fusion. `IsTerminator` gates DCE. Each trait-pass pair is a separate definitional relationship. No single trait predicts all pass behaviors.

TVM uses one 6-valued enum (`OpPattern`) that drives one behavior (fusion). It is closer to our setup—a single categorical coordinate predicting a behavioral outcome—but it predicts only one behavior, and its predictive accuracy has not been measured.

Futhark uses a SOAC classification that determines pairwise fusion legality. Like TVM, it predicts one behavior.

Blisp's DependencyClass predicts four behaviors simultaneously. The structural novelty is the one-to-many relationship: one coordinate, multiple independently-verified behavioral dimensions. The methodological novelty is treating this relationship as an empirical question rather than a definitional one.

6.2 The Naming Convention as Coordinate Encoding

The Blisp naming convention `SHF_{DEP}_{OBS}_{LIN}_{NAME}` deserves comment. It is tempting to dismiss it as a stylistic choice—a verbose way of naming operations. But the naming convention has a computational consequence: the Rust type system requires exhaustive matching over enum variants, which means that every subsystem that processes the IR must account for every variant. When the variant name encodes semantic coordinates, exhaustive matching becomes exhaustive semantic coverage.

This is not true of label-based systems. An MLIR pass can ignore a trait it does not check. A match arm over Blisp variant names cannot ignore a coordinate it does not handle—the compiler rejects incomplete matches. The naming convention is, in effect, a mechanism for encoding semantic coordinates at the type level, where the compiler enforces completeness.

6.3 Implications for Agent Systems

Modern tool-calling agent systems represent tools via natural-language descriptions. The agent must infer execution behavior from prose. This is equivalent to the "opaque identifier" condition in our representation comparison (section 4.7), which achieves 0% behavioral prediction.

If tools carried semantic coordinates, an agent could apply fixed rules to predict behavior without examples or training. The holdout experiment—where frozen rules predict the behavior of unseen operations at 100% accuracy—is a demonstration of this capability. An agent given coordinates for a tool it has never encountered can predict its fusion eligibility, warmup requirements, mask interaction, and optimization boundaries immediately.

We do not claim that all tool behaviors can be predicted from coordinates. We claim that the behaviors studied here—fusion, warmup, mask awareness, and optimization boundaries—are predictable, and that the prediction is empirically verified.

7 Limitations

Self-designed system. Both the coordinate system and the optimizer were designed by the same team. The 99.6% accuracy may reflect design discipline rather than an intrinsic property of time-series operations. The holdout test mitigates this concern (the prediction rules were frozen before holdout evaluation, and 5 holdout operations are hypothetical), but it does not eliminate it. External replication—testing whether DependencyClass predicts optimizer behavior in a system built by different people—is the most important direction for future work.

Domain-specific. The seven dependency-shape categories were developed for time-series operations. We do not know whether they transfer to other domains (image processing, graph computation, string manipulation). Generality claims require evidence that we do not yet have.

Small corpus. The corpus contains 61 operations (86 with the holdout). Larger operation sets may reveal additional exceptions beyond Keep_Al, requiring further coordinate refinements.

Single misprediction. The Keep_Al error shows that DependencyClass is not complete. A second coordinate (observation stream) is needed for full coverage. The paper presents a system with 99.6% accuracy, not 100%.

Four behaviors. We study four behavioral dimensions. Other optimizer decisions (cost estimation, memory allocation, parallelization strategy) may not be as strongly predicted by dependency shape. We do not claim completeness over all possible behavioral dimensions.

8 Related Work

Operation properties in compilers. Lattner et al. [1] describe MLIR's trait and interface system, which attaches properties to operations for use in optimization passes. Our work extends this by empirically measuring the predictive power of such properties rather than treating them as definitional. Chen et al. [2] describe TVM's `OpPattern` system, which is structurally closest to our approach (a single categorical coordinate driving fusion), but addresses only one behavioral dimension without empirical measurement of predictive power.

Array language classification. Henriksen et al. [3] classify operations in Futhark as SOACs and define fusion rules as a matrix over the classification. This is a definitional relationship (the classification defines fusion legality) rather than a measured predictive one.

Symbolic computation properties. Wolfram [4] defines attributes on symbolic operations (`Orderless`, `Flat`, `Listable`) that drive the pattern matcher and evaluator. These are mechanical gates, not statistical predictors.

Polyhedral analysis. Bondhugula et al. [5] characterize loop nests by their affine access patterns and use these to compute optimal tiling and fusion decisions. The access pattern is an exact characterization, not a categorical classification, and legality is proven rather than predicted. The polyhedral approach is complementary: it provides exact answers for the programs it can analyze, while our coordinate-based approach provides approximate (99.6%) answers for a broader class of operations including those with non-affine structure (recursive filters, exponential moving averages).

Machine learning for compiler optimization. Cummins et al. [6] and others use program features to predict optimization outcomes. These approaches extract features from program structure (instruction mix, loop depth, memory access patterns) rather than using declared semantic properties. The features are computed, not declared; and the predicted outcome is typically optimization *benefit* (speedup, cost), not optimization *legality* (fusable, mask-aware).

9 Conclusion

We have presented evidence that a single semantic coordinate—DependencyClass, a 7-valued classification of time-series operations by their data-dependency structure—predicts four independent optimizer behaviors with 99.6% accuracy and generalizes to unseen operations (100/100 on a 25-operation holdout, including 5 not yet implemented). The coordinate is not a passive label: it is consumed by four independently-developed subsystems in Blisp, functioning as a shared predictive object rather than a per-subsystem definitional gate.

The contribution is not Blisp itself. Blisp is a prototype architecture in which semantic coordinates are explicit, first-class, and operationally consumed, making them amenable to empirical measurement. The contribution is the measured relationship: explicit semantic coordinates can predict execution behavior, and the predictive power can be quantified using standard information-theoretic tools.

We see two directions for future work. First, external replication: testing whether Blisp's dependency-shape categories predict optimizer behavior in time-series systems built by different teams (pandas, kdb+, Apache Flink, Polars). If the same seven categories predict behavior across independently-designed systems, the finding is structural—a property of time-series operations, not of Blisp's design. If they do not, the finding is a local observation about disciplined engineering. Second, coordinate-augmented tool calling: testing whether exposing semantic coordinates to language-model agents reduces planning errors and enables reasoning about unseen tools, as the holdout experiment suggests.

The present paper establishes the empirical foundation: semantic coordinates can be predictive objects. Whether they are *generally* predictive objects is the question that follows.

References

[1] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *CGO*, 2021.

[2] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*, 2018.

[3] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *PLDI*, 2017.

[4] S. Wolfram. *The Mathematica Book*. Wolfram Media, 5th edition, 2003.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *PLDI*, 2008.

[6] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. P. O'Boyle, and H. Leather. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. In *CGO*, 2022.

A Full Corpus

The 61 operations used in the main experiment, with their DependencyClass assignments and ground-truth behaviors. Operation 60 (Keep_Al) is the single misprediction: predicted `mask_aware` = N, actual = Y (**bold**).

#	Operation	DC	EW	FB	Warmup	Mask
1	LOG	Elem	Y	N	None	N
2	EXP	Elem	Y	N	None	N
3	SQRT	Elem	Y	N	None	N
4	ABS	Elem	Y	N	None	N
5	SIGN	Elem	Y	N	None	N
6	INV	Elem	Y	N	None	N
7	CLAMP	Elem	Y	N	None	N
8	TANH_SAT	Elem	Y	N	None	N
9	ADD	Elem	Y	N	None	N
10	SUB	Elem	Y	N	None	N
11	MUL	Elem	Y	N	None	N
12	DIV	Elem	Y	N	None	N
13	GTR	Elem	Y	N	None	N
14	LSS	Elem	Y	N	None	N
15	GTE	Elem	Y	N	None	N
16	NEQ	Elem	Y	N	None	N
17	DLOG_OBS	Ptw	N	Y	Lag	Y
18	DLOG_OFS	Ptw	N	Y	Lag	Y
19	SHF_k	Ptw	N	Y	Lag	Y
20	SHF_FWD	Ptw	N	Y	Lag	Y
21	XMINUS_OBS	Ptw	N	Y	Lag	Y
22	XMINUS_OFS	Ptw	N	Y	Lag	Y
23	ROL_AVG	Win	N	Y	Window	Y
24	ROL_STD	Win	N	Y	Window	Y
25	ROL_AVG_MIN2	Win	N	Y	Window	Y
26	ROL_STD_MIN2	Win	N	Y	Window	Y
27	ROL_AVG_EXCL	Win	N	Y	Window	Y
28	ROL_STD_EXCL	Win	N	Y	Window	Y
29	ROL_AVG_M2_EXCL	Win	N	Y	Window	Y
30	ROL_STD_M2_EXCL	Win	N	Y	Window	Y
31	ROL_MAX	Win	N	Y	Window	Y
32	ROL_MIN	Win	N	Y	Window	Y
33	ROL_MAX_IND	Win	N	Y	Window	Y
34	ROL_MIN_IND	Win	N	Y	Window	Y

#	Operation	DC	EW	FB	Warmup	Mask
35	RSK_ADJ	Win	N	Y	Window	Y
36	RSK_ADJ_COEF	Win	N	Y	Window	Y
37	CUMSUM	Pfx	N	Y	Full	Y
38	RUN_SUM	Pfx	N	Y	Full	Y
39	EMA	Rec	N	Y	Full	Y
40	LOCF	Rec	N	Y	Full	Y
41	COV	Gbl	N	Y	None	N
42	COR	Gbl	N	Y	None	N
43	DET	Gbl	N	Y	None	N
44	ANN	Gbl	N	Y	None	N
45	EIG	Gbl	N	Y	None	N
46	FLAT	Gbl	N	Y	None	N
47	ZSCORE	Gbl	N	Y	None	N
48	RANK_ASC	Gbl	N	Y	None	N
49	RANK_DSC	Gbl	N	Y	None	N
50	WHERE	Gbl	N	Y	None	N
51	MEAN	Gbl	N	Y	None	N
52	MSK_WKE	Msk	N	Y	None	N
53	MSK_WKE_DEF	Msk	N	Y	None	N
54	CGREP	Sel	N	Y	None	N
55	CGREP_EQ	Sel	N	Y	None	N
56	COL_SELECT	Sel	N	Y	None	N
57	ALIGN	Sel	N	Y	None	N
58	ASOF_ALIGN	Sel	N	Y	None	N
59	SPR	Sel	N	Y	None	N
60	KEEP_AL	Sel	N	Y	None	**Y**
61	ROL_COR	Sel	N	Y	None	N

B Holdout Operations

The 25 operations used in the holdout experiment, with categories. All 100 predictions (25 operations × 4 behaviors) are correct.

Operation	Category	DC	All 4 correct
ADD	binary_arith	Elem	Y
SUB	binary_arith	Elem	Y
MUL	binary_arith	Elem	Y
DIV	binary_arith	Elem	Y
GTR	binary_comp	Elem	Y
LSS	binary_comp	Elem	Y
GTE	binary_comp	Elem	Y
NEQ	binary_comp	Elem	Y
SHF_GBL_MTX_COV	matrix	Gbl	Y
SHF_GBL_MTX_COR	matrix	Gbl	Y
SHF_GBL_MTX_DET	matrix	Gbl	Y
SHF_GBL_MTX_ANN	matrix	Gbl	Y
SHF_GBL_MTX_EIG	matrix	Gbl	Y
SHF_GBL_MTX_FLAT	matrix	Gbl	Y
SHF_WIN_NLN_MAX_IND	windowed	Win	Y
SHF_WIN_NLN_MIN_IND	windowed	Win	Y

Operation	Category	DC	All 4 correct
ZSCORE_AXIS	cross_axis	Gbl	Y
RANK_AXIS	cross_axis	Gbl	Y
SHF_GBL_VEC_WHERE	ternary	Gbl	Y
SHF_PTW_LIN_SPR	schema	Gbl	Y
SHF_WIN_NLN_MED	hypothetical	Win	Y
SHF_WIN_NLN_SKW	hypothetical	Win	Y
SHF_PFX_NLN_PROD	hypothetical	Pfx	Y
SHF_PFX_NLN_MAX	hypothetical	Pfx	Y
SHF_REC_NLN_EWM_SDV	hypothetical	Rec	Y

C Reproducibility

All experiments are implemented as automated tests in the Blisp repository:

Experiment	Test File	Tests
Main accuracy	`semantic_coordinate_information_theory.rs`	9
Holdout	`semantic_coordinate_holdout.rs`	1
All Paper 6	`artifacts/paper6/scripts/reproduce_all.sh`	62

Table 12: Reproducibility: every numerical claim is backed by an automated test.

To reproduce all results:

```
cargo test semantic_coordinate -- --nocapture
bash artifacts/paper6/scripts/reproduce_all.sh
```

Every numerical claim in this paper is produced by an automated test. No result depends on manual inspection or ad-hoc scripts.