Typing Discipline Selection for Object-Oriented Systems

A Formal Methodology with Empirical Validation

Abstract

We prove that shape-based typing disciplines—structural typing and duck typing—are categorically insufficient for systems requiring provenance tracking, MRO-based resolution, or type-identity dispatch. In greenfield development, where the architect controls the type hierarchy, shape-based typing is not a stylistic alternative to nominal typing. It is incorrect.

Shape-based typing is a retrofit concession: acceptable when the architect cannot control the inheritance hierarchy, never a design choice. This reframes typing discipline selection from subjective preference to objective decision procedure. Given requirements, the correct discipline is derivable—not chosen.

We validate through 13 case studies from a production bioimage analysis platform, formal theorems about Python's class system, and machine-checked proofs in Lean 4. Four metrics derived from the formal model (duck typing density, nominal typing ratio, provenance capability, resolution determinism) enable automated detection of violations. The methodology applies to any language with explicit inheritance: Python, Java, C#, Ruby.

---

1. Introduction

Practitioners face a recurring question: when should I use duck typing (`hasattr`, `getattr`, `try/except AttributeError`) versus nominal typing (`isinstance`, ABC contracts, explicit inheritance)? The Python community treats this as a style preference. It is not.

This paper proves the question has a definitive answer. **Duck typing and structural typing are shape-based**: they check what attributes or methods an object has. **Nominal typing is identity-based**: it checks what type an object is. When you control the type hierarchy (greenfield), identity-based typing is correct. Shape-based typing in this context is wrong—not suboptimal, wrong.

Shape-based typing exists for retrofit: integrating components you cannot modify, where you cannot mandate inheritance from your base classes. This is a concession to code outside your control. It is never a design choice. The concession is to practicality, not correctness.

We derive this methodology from formal theorems about Python's class system:

1. **The greenfield-retrofit distinction** (Theorem 3.4): Languages with explicit inheritance (`bases` axis) mandate nominal typing. Structural typing is valid only when `bases = []` universally.
2. **Complexity separation** (Theorem 4.3): Nominal typing achieves O(1) error localization; duck typing requires $\Omega(n)$ call-site inspection.
3. **Provenance impossibility** (Corollary 6.3): Duck typing cannot answer "which type provided this value?" because structurally equivalent objects are indistinguishable by definition. Machine-checked in Lean 4.

These theorems yield four measurable code quality metrics:

Metric	What it measures	Indicates
Duck typing density	`hasattr()` + `getattr()` + `try/except AttributeError` per KLOC	Retrofit patterns (acceptable) or discipline violations (problematic in greenfield)
Nominal typing ratio	`isinstance()` + ABC registrations per KLOC	Explicit type contracts
Provenance capability	Presence of "which type provided this" queries	System requires nominal typing
Resolution determinism	MRO-based dispatch vs runtime probing	O(1) vs Ω(n) error localization

The methodology is validated through 13 case studies from OpenHCS, a production bioimage analysis platform. The system's architecture exposed the formal necessity of nominal typing through patterns ranging from metaclass auto-registration to bidirectional type registries. A migration from duck typing to nominal contracts (PR #44) eliminated 47 scattered `hasattr()` checks and consolidated dispatch logic into explicit ABC contracts.

Contributions

1. **Methodology for typing discipline selection**: We formalize the greenfield-retrofit distinction—duck typing is retrofit tooling, nominal typing is greenfield tooling—and prove this is a correctness criterion, not a style choice (Theorem 3.4).
2. **Measurable code quality metrics**: We derive four metrics from the formal model (duck typing density, nominal typing ratio, provenance capability, resolution determinism) that enable automated detection of typing discipline violations.
3. **13 case studies demonstrating methodology application**: We analyze architectural patterns from a production system, showing how the methodology identifies which typing discipline is correct for each pattern. Measured outcomes include elimination of scattered `hasattr()` checks when migrating from duck typing to nominal contracts.
4. **Formal foundation**: We prove that Python's `type()` constructor constitutes a complete foundation for class-based systems (Theorem 2.1), that nominal typing achieves O(1) error localization versus duck typing's Ω(n) (Theorem 4.3), and that duck typing cannot provide provenance (Corollary 6.3, machine-checked in Lean 4).
5. **Recurring architectural patterns**: We identify six patterns that require nominal typing: metaclass auto-registration, bidirectional type registries, MRO-based priority resolution, runtime class generation with lineage tracking, descriptor protocol integration, and discriminated unions via `__subclasses__()`.

Empirical Context: OpenHCS

What it does: OpenHCS is a bioimage analysis platform. Pipelines are compiled before execution—errors surface at definition time, not after processing starts. The GUI and Python code are

interconvertible: design in GUI, export to code, edit, re-import. Changes to parent config propagate automatically to all child windows.

Why it matters for this paper: The system requires knowing *which type* provided a value, not just *what* the value is. Dual-axis resolution walks both the context hierarchy (global → plate → step) and the class hierarchy (MRO) simultaneously. Every resolved value carries provenance: (value, source_scope, source_type). This is only possible with nominal typing—duck typing cannot answer "which type provided this?"

Key architectural patterns (detailed in Section 5): - `@global_pipeline_config` decorator chain: one decorator spawns a 5-stage type transformation (Case Study 7) - Dual-axis resolver: MRO *is* the priority system—no custom priority function exists (Case Study 8) - Bidirectional type registries: single source of truth with `type()` identity as key (Case Study 13)

Decision Procedure, Not Preference

The contribution of this paper is not the theorems alone, but their consequence: typing discipline selection becomes a decision procedure. Given requirements, the discipline is derived.

Implications:

1. **Pedagogy.** Architecture courses should not teach "pick the style that feels Pythonic." They should teach how to derive the correct discipline from requirements. This is engineering, not taste.
2. **AI code generation.** LLMs can apply the decision procedure. "Given requirements R, apply Algorithm 1, emit code with the derived discipline" is an objective correctness criterion. The model either applies the procedure correctly or it does not.
3. **Language design.** Future languages could enforce discipline based on declared requirements. A `@requires_provenance` annotation could mandate nominal patterns at compile time.
4. **Ending debates.** "I prefer duck typing" is not a valid position when requirements include provenance. Preference is mathematically incorrect for the stated requirements. The procedure resolves the debate.

Scope: Absolute Claims

This paper makes absolute claims. We do not argue nominal typing is "preferred" or "more elegant." We prove:

1. **Shape-based typing cannot provide provenance.** Duck typing and structural typing check type *shape*—attributes, method signatures. Provenance requires type *identity*. Shape-based disciplines cannot provide what they do not track.
2. **In greenfield development, shape-based typing is wrong.** If the architect controls the type hierarchy, there is no reason to probe attributes instead of checking identity. The types are known. Checking shape discards information.
3. **Shape-based typing is a retrofit concession.** When integrating code you do not control, you cannot mandate inheritance from your base classes. Shape-based typing handles this case. It is a concession to external constraints—not a design choice, not correctness.

We do not claim all systems require provenance. We prove that systems requiring provenance cannot use shape-based typing. The requirements are the architect's choice; the discipline, given requirements, is derived.

Roadmap

Section 2 establishes preliminaries: Python's type system formalization and C3 linearization. Section 3 presents the greenfield-retrofit distinction—the core of the methodology. Section 4 formalizes complexity bounds (O(1) vs $\Omega(n)$) and information scattering. Section 5 demonstrates the methodology through 13 case studies, showing how to identify which typing discipline each pattern requires. Section 6 presents machine-checked verification in Lean 4, including the duck typing impossibility proof. Section 7 positions our work against prior research. Section 8 discusses limitations, appropriate use of structural typing, and derivable code quality metrics. Section 9 concludes with the methodology summary.

---

2. Preliminaries

2.1 Definitions

Definition 2.1 (Class). A class C is a triple (name, bases, namespace) where: - name $\in$ String — the identity of the class - bases $\in$ List[Class] — explicit inheritance declarations - namespace $\in$ Dict[String, Any] — attributes and methods

Definition 2.2 (Typing Discipline). A typing discipline T is a method for determining whether an object x satisfies a type constraint A.

Definition 2.3 (Nominal Typing). x satisfies A iff A $\in$ MRO(type(x)). The constraint is checked via explicit inheritance.

Definition 2.4 (Structural Typing). x satisfies A iff namespace(x) $\supseteq$ signature(A). The constraint is checked via method/attribute matching. In Python, `typing.Protocol` implements structural typing: a class satisfies a Protocol if it has matching method signatures, regardless of inheritance.

Definition 2.5 (Duck Typing). x satisfies A iff hasattr(x, m) returns True for each m in some implicit set M. The constraint is checked via runtime string-based probing.

Observation 2.1 (Shape-Based Typing). Structural typing and duck typing are both *shape-based*: they check what methods or attributes an object has, not what type it is. Nominal typing is *identity-based*: it checks the inheritance chain. This distinction is fundamental. Python's `Protocol`, TypeScript's interfaces, and Go's implicit interface satisfaction are all shape-based. ABCs with explicit inheritance are identity-based. The theorems in this paper prove shape-based typing cannot provide provenance—regardless of whether the shape-checking happens at compile time (structural) or runtime (duck).

Complexity distinction: While structural typing and duck typing are both shape-based, they differ critically in *when* the shape-checking occurs:

- **Structural typing** (Protocol): Shape-checking at *static analysis time* or *type definition time*. Complexity: O(k) where k = number of classes implementing the protocol.
- **Duck typing** (hasattr/getattr): Shape-checking at *runtime, per call site*. Complexity: $\Omega(n)$ where n = number of call sites.

This explains why structural typing (TypeScript interfaces, Go interfaces, Python Protocols) is considered superior to duck typing in practice: both are shape-based, but structural typing performs the checking once at compile/definition time, while duck typing repeats the checking at every usage site.

Critical insight: Even though structural typing has better complexity than duck typing (O(k) vs $\Omega(n)$), *both* are strictly dominated by nominal typing's O(1) error localization (Theorem 4.1). Nominal typing checks inheritance at the single class definition point—not once per implementing class (structural) or once per call site (duck).

2.2 The type() Theorem

Theorem 2.1 (Completeness). For any valid triple (name, bases, namespace), `type(name, bases, namespace)` produces a class C with exactly those properties.

Proof. By construction:

```
C = type(name, bases, namespace)
assert C.__name__ == name
assert C.__bases__ == bases
assert all(namespace[k] == getattr(C, k) for k in namespace)
```

The `class` statement is syntactic sugar for `type()`. Any class expressible via syntax is expressible via `type()`. ■

Theorem 2.2 (Semantic Minimality). The semantically minimal class constructor has arity 2: `type(bases, namespace)`.

Proof. - `bases` determines inheritance hierarchy and MRO - `namespace` determines attributes and methods - `name` is metadata; object identity distinguishes types at runtime - Each call to `type(bases, namespace)` produces a distinct object - Therefore name is not necessary for type semantics. ■

Theorem 2.3 (Practical Minimality). The practically minimal class constructor has arity 3: `type(name, bases, namespace)`.

Proof. The name string is required for: 1. **Debugging**: `repr(C)` → `<class '__main__.Foo'>` vs `<class '__main__.???'>` 2. **Serialization**: Pickling uses `__name__` to reconstruct classes 3. **Error messages**: "Expected Foo, got Bar" requires names 4. **Metaclass protocols**: `__init_subclass__`, registries key on `__name__`

Without name, the system is semantically complete but practically unusable. ■

Definition 2.6 (The Two-Axis Semantic Core). The semantic core of Python's class system is: - **bases**: inheritance relationships (→ MRO, nominal typing) - **namespace**: attributes and methods (→ behavior, structural typing)

The `name` axis is orthogonal to both and carries no semantic weight.

Theorem 2.4 (Orthogonality of Semantic Axes). The `bases` and `namespace` axes are orthogonal.

Proof. Independence: - Changing bases does not change namespace content (only resolution order for inherited methods) - Changing namespace does not change bases or MRO

The factorization (bases, namespace) is unique. ■

Corollary 2.5. The semantic content of a class is fully determined by (bases, namespace). Two classes with identical bases and namespace are semantically equivalent, differing only in object identity.

2.3 C3 Linearization (Prior Work)

Theorem 2.6 (C3 Optimality). C3 linearization is the unique algorithm satisfying: 1. **Monotonicity:** If A precedes B in linearization of C, and C' extends C, then A precedes B in linearization of C' 2. **Local precedence:** A class precedes its parents in its own linearization 3. **Consistency:** Linearization respects all local precedence orderings

Proof. See Barrett et al. (1996), "A Monotonic Superclass Linearization for Dylan." ■

Corollary 2.7. Given bases, MRO is deterministically derived. There is no configuration; there is only computation.

2.4 Abstract Class System Model

We formalize class systems independently of any specific language. This establishes that our theorems apply to **any** language with explicit inheritance, not just Python.

2.4.1 The Three-Axis Model **Definition 2.7 (Abstract Class System).** A class system is a tuple (N, B, S) where: - N: Name — the identifier for a type - B: Bases — the set of explicitly declared parent types (inheritance) - S: Namespace — the set of (attribute, value) pairs defining the type's interface

Definition 2.8 (Class Constructor). A class constructor is a function:

$$\text{class} : N \times \mathcal{P}(T) \times S \to T$$

where T is the universe of types, taking a name, a set of base types, and a namespace, returning a new type.

Language instantiations:

Language	Name	Bases	Namespace	Constructor Syntax
Python	`str`	`tuple[type]`	`dict[str, Any]`	`type(name, bases, namespace)`
Java	`String`	`Class<?>`	method/field declarations	`class Name extends Base { ... }`
C#	`string`	`Type`	member declarations	`class Name : Base { ... }`
Ruby	`Symbol`	`Class`	method definitions	`class Name < Base; end`
TypeScript	`string`	`Function`	property declarations	`class Name extends Base { ... }`

Definition 2.9 (Reduced Class System). A class system is *reduced* if $B = \emptyset$ for all types (no inheritance). Examples: Go (structs only), C (no classes), JavaScript ES5 (prototype-based, no `class` keyword).

2.4.2 Typing Disciplines as Axis Projections **Definition 2.10 (Shape-Based Typing).** A typing discipline is *shape-based* if type compatibility is determined solely by S (namespace):

$$\text{compatible}_{\text{shape}}(x, T) \iff S(\text{type}(x)) \supseteq S(T)$$

Shape-based typing projects out the B axis entirely. It cannot distinguish types with identical namespaces.

Definition 2.11 (Nominal Typing). A typing discipline is *nominal* if type compatibility requires identity in the inheritance hierarchy:

$$\text{compatible}_{\text{nominal}}(x, T) \iff T \in \text{ancestors}(\text{type}(x))$$

where $\text{ancestors}(C) = \{C\} \cup \bigcup_{P \in B(C)} \text{ancestors}(P)$ (transitive closure over B).

2.4.3 Provenance as MRO Query **Definition 2.12 (Provenance Query).** A provenance query asks: "Given object x and attribute a, which type $T \in \text{MRO}(\text{type}(x))$ provided the value of a?"

Theorem 2.7 (Provenance Requires MRO). Provenance queries require access to MRO, which requires access to B.

Proof. MRO is defined as a linearization over ancestors, which is the transitive closure over B. Without B, MRO is undefined. Without MRO, provenance queries cannot be answered. ■

Corollary 2.8 (Shape-Based Typing Cannot Provide Provenance). Shape-based typing cannot answer provenance queries.

Proof. By Definition 2.10, shape-based typing uses only S. By Theorem 2.7, provenance requires B. Shape-based typing has no access to B. Therefore shape-based typing cannot provide provenance. ■

2.4.4 Cross-Language Instantiation **Table 2.1: Cross-Language Instantiation of the (N, B, S) Model**

Language	N (Name)	B (Bases)	S (Namespace)	Type System
Python	`type(x).__name__`	`__bases__`, `__mro__`	`__dict__`, `dir()`	Nominal
Java	`getClass().getName()`	`getSuperclass()`, `getInterfaces()`	`getDeclaredMethods()`	Nominal
Ruby	`obj.class.name`	`ancestors` (C3 linearization)	`methods`, `instance_variables`	Nominal
C#	`GetType().Name`	`BaseType`, `GetInterfaces()`	`GetProperties()`, `GetMethods()`	Nominal

All four languages provide **runtime access to all three axes**. The critical difference lies in which axes the **type system** inspects.

2.5 The Axis Lattice Metatheorem

The three-axis model (N, B, S) induces a lattice of typing disciplines. Each discipline is characterized by which axes it inspects:

Axis Subset	Discipline	Example
$\emptyset$	Untyped	Accept all
$\{N\}$	Named-only	Type aliases
$\{S\}$	Pure structural	Interface matching
$\{N, S\}$	Shape-based	Duck typing, Protocols
$\{N, B, S\}$	Nominal	ABCs, isinstance

Theorem 2.9 (Axis Lattice Dominance). For any axis subsets $A \subseteq B \subseteq \{N, B, S\}$, the capabilities of discipline using A are a subset of capabilities of discipline using B:

$$\text{capabilities}(A) \subseteq \text{capabilities}(B)$$

Proof. Each axis enables specific capabilities: - N: Type naming, aliasing - B: Provenance, identity, enumeration, conflict resolution - S: Interface checking

A discipline using subset A can only employ capabilities enabled by axes in A. Adding an axis to A adds capabilities but removes none. Therefore the capability sets form a monotonic lattice under subset inclusion. ■

Corollary 2.10 (Bases Axis Primacy). The Bases axis B is the source of all strict dominance. Specifically: provenance, type identity, subtype enumeration, and conflict resolution all require B. Any discipline that discards B forecloses these capabilities.

Theorem 2.11 (Strict Dominance — Abstract). In any class system with $B \neq \emptyset$, nominal typing strictly dominates shape-based typing.

Proof. Let $\mathcal{C}_{\text{shape}}$ = capabilities of shape-based typing. Let $\mathcal{C}_{\text{nominal}}$ = capabilities of nominal typing.

Shape-based typing can check interface satisfaction: $S(\text{type}(x)) \supseteq S(T)$.

Nominal typing can: 1. Check interface satisfaction (equivalent to shape-based) 2. Check type identity: $T \in \text{ancestors}(\text{type}(x))$ — **impossible for shape-based** 3. Answer provenance queries — **impossible for shape-based** (Corollary 2.8) 4. Enumerate subtypes — **impossible for shape-based** 5. Use type as dictionary key — **impossible for shape-based**

Therefore $\mathcal{C}_{\text{shape}} \subset \mathcal{C}_{\text{nominal}}$ (strict subset). In a class system with $B \neq \emptyset$, both disciplines are available. Choosing shape-based typing forecloses capabilities for zero benefit. ■

2.5.1 The Decision Procedure Given a language L and development context C:

```
FUNCTION select_typing_discipline(L, C):
    IF L has no inheritance syntax (B = $\emptyset$):
        RETURN structural  # Theorem 3.1: correct for reduced systems

    IF C is retrofit (cannot modify type definitions):
        RETURN structural  # Concession to external constraints
```

```
IF C is greenfield (architect controls types):
    RETURN nominal  # Theorem 2.11: strict dominance
```

This is a **decision procedure**, not a preference. The output is determined by the inputs.

---

3. The Greenfield Distinction

Thought experiment: What if `type()` only took namespace?

Given that the semantic core is (bases, namespace), what if we further reduce to just namespace?

```python
# Hypothetical minimal class constructor
def type_minimal(namespace: dict) -> type:
    """Create a class from namespace only."""
    return type("", (), namespace)
```

Definition 3.1 (Namespace-Only System). A namespace-only class system is one where: - Classes are characterized entirely by their namespace (attributes/methods) - No explicit inheritance mechanism exists (bases axis absent)

Theorem 3.1 (Structural Typing Is Correct for Namespace-Only Systems).

In a namespace-only system, structural typing is the unique correct typing discipline.

Proof. 1. Let A and B be classes in a namespace-only system 2. A $\equiv$ B iff namespace(A) = namespace(B) (by definition of namespace-only) 3. Structural typing checks: namespace(x) $\supseteq$ signature(T) 4. This is the only information available for type checking 5. Therefore structural typing is correct and complete. ■

Corollary 3.2 (Go's Design Is Consistent). Go has no inheritance. Interfaces are method sets. Structural typing is correct for Go.

Corollary 3.3 (TypeScript's Design Is Consistent). TypeScript classes are structural. No runtime inheritance hierarchy is checked. Structural typing is correct for TypeScript's type system.

The Critical Observation (Semantic Axes):

System	Semantic Axes	Correct Discipline
Namespace-only	`(namespace)`	Structural
Full Python	`(bases, namespace)`	Nominal

The `name` axis is metadata in both cases—it doesn't affect which typing discipline is correct.

Theorem 3.4 (Bases Mandates Nominal). The presence of a `bases` axis in the class system mandates nominal typing for greenfield development.

Proof. 1. Python's class system has two semantic axes: (bases, namespace) 2. `bases` encodes explicit inheritance relationships forming a DAG 3. C3 linearization derives deterministic MRO from `bases` 4. Shape-based typing (structural, duck) checks only namespace—ignores `bases` entirely 5. Therefore shape-based typing discards semantic information present in the system

The key question: Is the discarded information necessary?

Consider provenance tracking: "Which type in the MRO provided this value?"

6. Provenance requires distinguishing types at different MRO positions
7. Shape-based typing sees structurally identical types as indistinguishable (by definition)
8. Therefore shape-based typing cannot answer provenance queries (Corollary 6.3)
9. Nominal typing uses `bases` via isinstance(x, A), which checks MRO position
10. Therefore nominal typing can answer provenance queries

The `bases` axis creates a semantic distinction (MRO position) that shape-based typing cannot represent. Systems requiring this distinction cannot use shape-based typing. Since greenfield architects control whether to use `bases`, and nominal typing is the only discipline that uses `bases`, the presence of `bases` mandates nominal typing for systems requiring provenance. ■

Theorem 3.5 (Strict Dominance in Greenfield). In greenfield development, nominal typing strictly dominates shape-based typing: nominal provides all capabilities of shape-based typing plus additional capabilities, at equal declaration cost.

Proof. Consider Python's concrete implementations: - Shape-based: `typing.Protocol` (structural typing) - Nominal: Abstract Base Classes (ABCs)

Let S = capabilities provided by Protocol, N = capabilities provided by ABCs.

What Protocols provide: 1. Interface enforcement via method signature matching 2. Type checking at static analysis time (mypy, pyright) 3. No runtime isinstance() check (by default)

What ABCs provide: 1. Interface enforcement via `@abstractmethod` (equivalent to Protocol) 2. Type checking at static analysis time (equivalent to Protocol) 3. **Type identity via isinstance()** (Protocol cannot provide this) 4. **Provenance tracking via MRO position** (Protocol cannot provide this) 5. **Exhaustive enumeration via `__subclasses__()`** (Protocol cannot provide this) 6. **Type-as-dictionary-key via type() identity** (Protocol cannot provide this) 7. **Runtime enforcement at instantiation** (Protocol only checks statically)

Therefore S $\subset$ N (strict subset). Both require explicit type declarations. The declaration cost is equivalent: one class definition per interface. Therefore, nominal typing provides strictly more capabilities at equal or lower cost (earlier failure). ■

Corollary 3.6 (Greenfield Incorrectness). In greenfield development, using shape-based typing instead of nominal typing is not suboptimal—it is incorrect.

Proof. By Theorem 3.5, nominal typing strictly dominates shape-based typing. Choosing a strictly dominated option when the superior option is available at equal cost is definitionally incorrect. ■

3.7 The Absolute Claim

Claim (Typing Discipline Correctness). In any programming language with explicit inheritance syntax, shape-based typing (structural typing, duck typing, Protocol-based typing) is **incorrect** for greenfield development. Nominal typing is **correct**. This is not a preference, recommendation, or tradeoff. It is a mathematical fact derivable from the structure of class systems.

Proof. By Theorem 2.11 (Strict Dominance), nominal typing provides all capabilities of shape-based typing plus additional capabilities (provenance, type identity, subtype enumeration, type-as-key).

By Corollary 2.10, in greenfield development where the architect controls type definitions, choosing the strictly dominated option is incorrect. ■

What "incorrect" means: 1. **Information-theoretic**: Shape-based typing discards the B axis. Discarding available information without compensating benefit is suboptimal by definition. 2. **Capability-theoretic**: Shape-based typing forecloses capabilities that nominal typing provides. Foreclosing capabilities for zero benefit is incorrect. 3. **Decision-theoretic**: Given the choice between two options where one strictly dominates, choosing the dominated option is irrational.

3.8 Information-Theoretic Completeness

A potential objection to Theorem 3.5 is that our capability enumeration is arbitrary. This section proves the enumeration is **derived from information structure**, not chosen.

Definition 3.10 (Query). A *query* is a predicate $q : \text{Type} \to \text{Bool}$ that a typing discipline can evaluate.

Definition 3.11 (Shape-Respecting Query). A query q is *shape-respecting* if for all types A, B with $S(A) = S(B)$:

$$q(A) = q(B)$$

That is, shape-equivalent types cannot be distinguished by q.

Theorem 3.12 (Capability Gap Characterization). Let ShapeQueries be the set of all shape-respecting queries, and let AllQueries be the set of all queries. If there exist types $A \neq B$ with $S(A) = S(B)$, then:

$$\text{ShapeQueries} \subsetneq \text{AllQueries}$$

Proof. The identity query $\text{isA}(T) := (T = A)$ is in AllQueries but not ShapeQueries, because isA(A) = true but isA(B) = false despite $S(A) = S(B)$. ■

Corollary 3.13 (Derived Capability Set). The capability gap between shape-based and nominal typing is **exactly** the set of queries that depend on the Bases axis:

$$\text{Capability Gap} = \{q \mid \exists A, B.\ S(A) = S(B) \wedge q(A) \neq q(B)\}$$

This is not an enumeration—it's a **characterization**. Our listed capabilities (provenance, identity, enumeration, conflict resolution) are instances of this set, not arbitrary choices.

Information-Theoretic Interpretation: Information theory tells us that discarding information forecloses queries that depend on that information. The Bases axis contains information about inheritance relationships. Shape-based typing discards this axis. Therefore, any query that depends on inheritance—provenance, identity, enumeration, conflict resolution—is foreclosed. This is not our claim; it's a mathematical necessity.

---

4. Core Theorems

4.1 The Error Localization Theorem

Definition 4.1 (Error Location). Let E(T) be the number of source locations that must be inspected to find all potential violations of a type constraint under discipline T.

Theorem 4.1 (Nominal Complexity). E(nominal) = O(1).

Proof. Under nominal typing, constraint "x must be an A" is satisfied iff type(x) inherits from A. This property is determined at class definition time, at exactly one location: the class definition of type(x). If the class does not list A in its bases (transitively), the constraint fails. One location. ■

Theorem 4.2 (Structural Complexity). E(structural) = O(k) where k = number of classes.

Proof. Under structural typing, constraint "x must satisfy interface A" requires checking that type(x) implements all methods in signature(A). This check occurs at each class definition. For k classes, O(k) locations. ■

Theorem 4.3 (Duck Typing Complexity). E(duck) = Ω(n) where n = number of call sites.

Proof. Under duck typing, constraint "x must have method m" is encoded as `hasattr(x, "m")` at each call site. There is no central declaration. For n call sites, each must be inspected. Lower bound is Ω(n). ■

Corollary 4.4 (Strict Dominance). Nominal typing strictly dominates duck typing: E(nominal) = O(1) < Ω(n) = E(duck) for all n > 1.

4.2 The Information Scattering Theorem

Definition 4.2 (Constraint Encoding Locations). Let I(T, c) be the set of source locations where constraint c is encoded under discipline T.

Theorem 4.5 (Duck Typing Scatters). For duck typing, |I(duck, c)| = O(n) where n = call sites using constraint c.

Proof. Each `hasattr(x, "method")` call independently encodes the constraint. No shared reference. Constraints scale with call sites. ■

Theorem 4.6 (Nominal Typing Centralizes). For nominal typing, |I(nominal, c)| = O(1).

Proof. Constraint c = "must inherit from A" is encoded once: in the ABC/Protocol definition of A. All `isinstance(x, A)` checks reference this single definition. ■

Corollary 4.7 (Maintenance Entropy). Duck typing maximizes maintenance entropy; nominal typing minimizes it.

4.3 Empirical Demonstration

The theoretical complexity bounds in Theorems 4.1-4.3 are demonstrated empirically in Section 5, Case Study 1 (WellFilterConfig hierarchy). Two classes with identical structure but different nominal identities require O(1) disambiguation under nominal typing but Ω(n) call-site inspection under duck typing. Case Study 5 provides measured outcomes: migrating from duck to nominal typing reduced error localization complexity from scattered `hasattr()` checks across 47 call sites to centralized ABC contract validation at a single definition point.

---

5. Case Studies: Applying the Methodology

5.1 Empirical Validation Strategy

Addressing the "n=1" objection: A potential criticism is that our case studies come from a single codebase (OpenHCS). We address this in three ways:

First: 13 independent decisions. OpenHCS made 13 independent architectural decisions, each of which could have gone structural. All 13 went nominal. Each decision is a data point. The probability that all 13 would go nominal by chance, if structural were equally valid, is $2^{-13} \approx 0.01\%$.

Second: Case studies are theorem instantiations. Table 5.1 links each case study to the theorem it validates. These are not arbitrary examples—they are empirical instantiations of theoretical predictions. The theory predicts that systems requiring provenance will use nominal typing; the case studies confirm this prediction.

Third: Falsifiable predictions. Section 9.2 provides explicit predictions for Django, Spring, Rails, and Go. If these predictions are wrong, our theory is falsified. This is the scientific method: make predictions, test them.

The validation structure:

Level	What it provides	Status
Formal proofs	Mathematical necessity	Complete (Lean, 758 lines, 0 `sorry`)
OpenHCS case studies	Existence proof	13 patterns documented
Cross-language predictions	Falsifiability	Section 9.2

OpenHCS is a bioimage analysis platform for high-content screening microscopy. The system was designed from the start with explicit commitment to nominal typing, exposing the consequences of this architectural decision through 13 distinct patterns. These case studies demonstrate the methodology in action: for each pattern, we identify whether it requires provenance tracking, MRO-based resolution, or type identity as dictionary keys—all indicators that nominal typing is mandatory per the formal model.

Duck typing fails for all 13 patterns because they fundamentally require **type identity** rather than structural compatibility. Configuration resolution needs to know *which type* provided a value (provenance tracking, Corollary 6.3). MRO-based priority needs inheritance relationships preserved (Theorem 3.4). Metaclass registration needs types as dictionary keys (type identity as hash). These requirements are not implementation details—they are architectural necessities proven impossible under duck typing's structural equivalence axiom.

The 13 studies demonstrate four pattern taxonomies: (1) **type discrimination** (WellFilterConfig hierarchy), (2) **metaclass registration** (AutoRegisterMeta, GlobalConfigMeta, DynamicInterfaceMeta), (3) **MRO-based resolution** (dual-axis resolver, @global_pipeline_config chain), and (4) **bidirectional lookup** (lazy $\leftrightarrow$ base type registries). Table 5.2 summarizes how each pattern fails under duck typing and what nominal mechanism enables it.

Table 5.1: Case Studies as Theorem Validation

Study	Pattern	Validates Theorem	Validation Type
1	Type discrimination	Theorem 3.4 (Bases Mandates Nominal)	MRO position distinguishes structurally identical types
2	Discriminated unions	Theorem 3.5 (Strict Dominance)	`__subclasses__()` provides exhaustiveness
3	Converter dispatch	Theorem 4.1 (O(1) Complexity)	`type()` as dict key vs O(n) probing
4	Polymorphic config	Corollary 6.3 (Provenance Impossibility)	ABC contracts track provenance
5	Architecture migration	Theorem 4.4 (Fail-Loud)	Definition-time vs runtime failure
6	Auto-registration	Theorem 3.5 (Strict Dominance)	`__init_subclass__` hook
7	Type transformation	Corollary 6.3 (Provenance Impossibility)	5-stage `type()` chain tracks lineage
8	Dual-axis resolution	Theorem 3.4 (Bases Mandates Nominal)	Scope × MRO product requires MRO
9	Custom isinstance	Theorem 3.5 (Strict Dominance)	`__instancecheck__` override
10	Dynamic interfaces	Theorem 3.5 (Strict Dominance)	Metaclass-generated ABCs
11	Framework detection	Theorem 4.1 (O(1) Complexity)	Sentinel type vs module probing
12	Method injection	Corollary 6.3 (Provenance Impossibility)	`type()` namespace manipulation
13	Bidirectional lookup	Theorem 4.1 (O(1) Complexity)	Single registry with `type()` keys

Table 5.2: Comprehensive Case Study Summary

Study	Pattern	Duck Failure Mode	Nominal Mechanism
1	Type discrimination	Structural equivalence	`isinstance()` + MRO position
2	Discriminated unions	No exhaustiveness check	`__subclasses__()` enumeration
3	Converter dispatch	O(n) attribute probing	`type()` as dict key
4	Polymorphic config	No interface guarantee	ABC contracts
5	Architecture migration	Fail-silent at runtime	Fail-loud at definition
6	Auto-registration	No type identity	`__init_subclass__` hook

Study	Pattern	Duck Failure Mode	Nominal Mechanism
7	Type transformation	Cannot track lineage	5-stage `type()` chain
8	Dual-axis resolution	No scope × MRO product	Registry + MRO traversal
9	Custom isinstance	Impossible	`__instancecheck__` override
10	Dynamic interfaces	No interface identity	Metaclass-generated ABCs
11	Framework detection	Module probing fragile	Sentinel type in registry
12	Method injection	No target type	`type()` namespace manipulation
13	Bidirectional lookup	Two dicts, sync bugs	Single registry, `type()` keys

5.2 Case Study 1: Structurally Identical, Semantically Distinct Types

```python
@dataclass(frozen=True)
class WellFilterConfig:
    """Pipeline-level well filtering."""
    well_filter: Optional[Union[List[str], str, int]] = None
    well_filter_mode: WellFilterMode = WellFilterMode.INCLUDE

@dataclass(frozen=True)
class StepWellFilterConfig(WellFilterConfig):
    """Step-level well filtering."""
    pass  # Structurally identical!
```

These classes are structurally identical but participate in different inheritance hierarchies. The MRO position determines resolution priority in OpenHCS's dual-axis configuration system:

Context hierarchy: Global → Pipeline → Step **MRO inheritance:** `StepMaterializationConfig $\rightarrow$ StepWellFilterConfig $\rightarrow$ PathPlanningConfig $\rightarrow$ WellFilterConfig`

When resolving `well_filter` on a step, the system walks this MRO. The *position* of `StepWellFilterConfig` vs `WellFilterConfig` in the chain determines which value is returned—structurally identical types at different MRO positions resolve to different values based on scope context.

Under duck typing, both classes have identical attributes (`well_filter`, `well_filter_mode`). There is no way to distinguish them. The system cannot answer "is this the pipeline-level config or step-level config?"—both have the same shape. Nominal typing provides `type(config) is StepWellFilterConfig` as an O(1) check (Theorem 4.1), while duck typing would require $\Omega(n)$ inspection of context metadata not present in the object itself.

Pattern (Table 5.1, Row 1): Type discrimination via MRO position. Demonstrates Theorem 4.3 (O(1) vs $\Omega(n)$ complexity) and serves as the canonical example of structural equivalence failing

to capture semantic distinctions.

5.3 Case Study 2: Discriminated Unions via subclasses()

OpenHCS's parameter UI needs to dispatch widget creation based on parameter type structure: `Optional[Dataclass]` parameters need checkboxes, direct `Dataclass` parameters are always visible, and primitive types use simple widgets. The challenge: how does the system enumerate all possible parameter types to ensure exhaustive handling?

```
@dataclass
class OptionalDataclassInfo(ParameterInfoBase):
    widget_creation_type: str = "OPTIONAL_NESTED"

    @staticmethod
    def matches(param_type: Type) -> bool:
        return is_optional(param_type) and is_dataclass(inner_type(param_type))

@dataclass
class DirectDataclassInfo(ParameterInfoBase):
    widget_creation_type: str = "NESTED"

    @staticmethod
    def matches(param_type: Type) -> bool:
        return is_dataclass(param_type)

@dataclass
class GenericInfo(ParameterInfoBase):
    @staticmethod
    def matches(param_type: Type) -> bool:
        return True  # Fallback
```

The factory uses `ParameterInfoBase.__subclasses__()` to enumerate all registered variants at runtime. This provides exhaustiveness: adding a new parameter type (e.g., `EnumInfo`) automatically extends the dispatch table without modifying the factory. Duck typing has no equivalent—there's no way to ask "what are all the types that have a `matches()` method?"

Structural typing would require manually maintaining a registry list. Nominal typing provides it for free via inheritance tracking. The dispatch is O(1) after the initial linear scan to find the matching subclass.

Pattern (Table 5.1, Row 2): Discriminated union enumeration. Demonstrates how nominal identity enables exhaustiveness checking that duck typing cannot provide.

5.4 Case Study 3: MemoryTypeConverter Dispatch

```
# 6 converter classes auto-generated at module load
_CONVERTERS = {
    mem_type: type(
        f"{mem_type.value.capitalize()}Converter",  # name
        (MemoryTypeConverter,),                      # bases
```

```
        _TYPE_OPERATIONS[mem_type]                    # namespace
    )()
    for mem_type in MemoryType
}

def convert_memory(data, source_type: str, target_type: str, gpu_id: int):
    source_enum = MemoryType(source_type)
    converter = _CONVERTERS[source_enum]  # O(1) lookup by type
    method = getattr(converter, f"to_{target_type}")
    return method(data, gpu_id)
```

This generates `NumpyConverter`, `CupyConverter`, `TorchConverter`, `TensorflowConverter`, `JaxConverter`, `PyclesperantoConverter`—all with identical method signatures (`to_numpy()`, `to_cupy()`, etc.) but completely different implementations.

The nominal type identity created by `type()` allows using converters as dict keys in `_CONVERTERS`. Duck typing would see all converters as structurally identical (same method names), making O(1) dispatch impossible. The system would need to probe each converter with hasattr or maintain a parallel string-based registry.

Pattern (Table 5.1, Row 3): Factory-generated types as dictionary keys. Demonstrates Theorem 4.1 (O(1) dispatch) and the necessity of type identity for efficient lookup.

5.5 Case Study 4: Polymorphic Configuration

The streaming subsystem supports multiple viewers (Napari, Fiji) with different port configurations and backend protocols. How should the orchestrator determine which viewer config is present without fragile attribute checks?

```
class StreamingConfig(StreamingDefaults, ABC):
    @property
    @abstractmethod
    def backend(self) -> Backend: pass

# Factory-generated concrete types
NapariStreamingConfig = create_streaming_config(
    viewer_name='napari', port=5555, backend=Backend.NAPARI_STREAM)
FijiStreamingConfig = create_streaming_config(
    viewer_name='fiji', port=5565, backend=Backend.FIJI_STREAM)

# Orchestrator dispatch
if isinstance(config, StreamingConfig):
    registry.get_or_create_tracker(config.port, config.viewer_type)
```

The codebase documentation explicitly contrasts approaches:

> **Old:** `hasattr(config, 'napari_port')` — fragile (breaks if renamed), no type checking **New:** `isinstance(config, NapariStreamingConfig)` — type-safe, explicit

Duck typing couples the check to attribute names (strings), creating maintenance fragility. Renaming a field breaks all `hasattr()` call sites. Nominal typing couples the check to type identity, which is

refactoring-safe.

Pattern (Table 5.1, Row 4): Polymorphic dispatch with interface guarantees. Demonstrates how nominal ABC contracts provide fail-loud validation that duck typing's fail-silent probing cannot match.

5.6 Case Study 5: Migration from Duck to Nominal Typing (PR #44)

PR #44 ("UI Anti-Duck-Typing Refactor", 90 commits, 106 files, +22,609/-7,182 lines) migrated OpenHCS's UI layer from duck typing to nominal ABC contracts. The measured architectural changes:

Before (duck typing): - ParameterFormManager: 47 `hasattr()` dispatch points scattered across methods - CrossWindowPreviewMixin: attribute-based widget probing throughout - Dispatch tables: string attribute names mapped to handlers

After (nominal typing): - ParameterFormManager: single `AbstractFormWidget` ABC with explicit contracts - CrossWindowPreviewMixin: explicit widget protocols - Dispatch tables: eliminated — replaced by `isinstance()` + method calls

Architectural transformation:

```
# BEFORE: Duck typing dispatch (scattered across 47 call sites)
if hasattr(widget, 'isChecked'):
    return widget.isChecked()
elif hasattr(widget, 'currentText'):
    return widget.currentText()
# ... 45 more cases

# AFTER: Nominal ABC (single definition point)
class AbstractFormWidget(ABC):
    @abstractmethod
    def get_value(self) -> Any: pass

# Error detection: attribute typos caught at import time, not user interaction time
```

The migration eliminated fail-silent bugs where missing attributes returned `None` instead of raising exceptions. Type errors now surface at class definition time (when ABC contract is violated) rather than at user interaction time (when attribute access fails silently).

Pattern (Table 5.1, Row 5): Architecture migration from fail-silent duck typing to fail-loud nominal contracts. Demonstrates measured reduction in error localization complexity (Theorem 4.3): from $\Omega(47)$ scattered hasattr checks to $O(1)$ centralized ABC validation.

5.7 Case Study 6: AutoRegisterMeta

Pattern: Metaclass-based auto-registration uses type identity as the registry key. At class definition time, the metaclass registers each concrete class (skipping ABCs) in a type-keyed dictionary.

```
class AutoRegisterMeta(ABCMeta):
    def __new__(mcs, name, bases, attrs, registry_config=None):
        new_class = super().__new__(mcs, name, bases, attrs)
```

```
        # Skip abstract classes (nominal check via __abstractmethods__)
        if getattr(new_class, '__abstractmethods__', None):
            return new_class

        # Register using type as value
        key = mcs._get_registration_key(name, new_class, registry_config)
        registry_config.registry_dict[key] = new_class
        return new_class

# Usage: Define class $\rightarrow$ auto-registered
class ImageXpressHandler(MicroscopeHandler, metaclass=MicroscopeHandlerMeta):
    _microscope_type = 'imagexpress'
```

This pattern is impossible with duck typing because: (1) type identity is required as dict values—duck typing has no way to reference "the type itself" distinct from instances, (2) skipping abstract classes requires checking `__abstractmethods__`, a class-level attribute inaccessible to duck typing's instance-level probing, and (3) inheritance-based key derivation (extracting "imagexpress" from "ImageXpressHandler") requires class name access.

The metaclass ensures exactly one handler per microscope type. Attempting to define a second `ImageXpressHandler` raises an exception at import time. Duck typing's runtime checks cannot provide this guarantee—duplicates would silently overwrite.

Pattern (Table 5.1, Row 6): Auto-registration with type identity. Demonstrates that metaclasses fundamentally depend on nominal typing to distinguish classes from instances.

5.8 Case Study 7: Five-Stage Type Transformation

The `@global_pipeline_config` decorator chain demonstrates nominal typing's power for systematic type manipulation. Starting from a base config, one decorator invocation spawns a 5-stage type transformation that generates lazy companion types, injects fields into parent configs, and maintains bidirectional registries.

Stage 1: `@auto_create_decorator` marks `GlobalPipelineConfig` with `_is_global_config = True` and creates the decorator itself via `setattr(module, 'global_pipeline_config', decorator)`.

Stage 2: `@global_pipeline_config(inherit_as_none=True)` on `PathPlanningConfig` triggers lazy type generation: `type("LazyPathPlanningConfig", (PathPlanningConfig, LazyDataclass), namespace)` where namespace contains all fields with `default=None`.

Stage 3: Descriptor protocol integration via `__set_name__` injects fields into parent configs. When `Pipeline` defines `path_planning: LazyPathPlanningConfig`, the descriptor automatically adds `path_planning` to `GlobalPipelineConfig` with `default_factory=LazyPathPlanningConfig`.

Stage 4: Bidirectional registries link lazy $\leftrightarrow$ base types: `_lazy_to_base[LazyPathPlanningConfig] = PathPlanningConfig` and `_base_to_lazy[PathPlanningConfig] = LazyPathPlanningConfig`. Normalization uses these at resolution time.

Stage 5: MRO-based resolution walks `type(config).__mro__`, normalizing each type via registry lookup. The sourceType in `(value, scope, sourceType)` carries provenance that duck typing

cannot provide.

This 5-stage chain is single-stage generation (not nested metaprogramming). It respects Veldhuizen's (2006) bounds: full power without complexity explosion. The lineage tracking (which lazy type came from which base) is only possible with nominal identity—structurally equivalent types would be indistinguishable.

Pattern (Table 5.1, Row 7): Type transformation with lineage tracking. Demonstrates the limits of what duck typing can express: runtime type generation requires `type()`, which returns nominal identities.

5.9 Case Study 8: Dual-Axis Resolution Algorithm

```
def resolve_field_inheritance(obj, field_name, scope_stack):
    mro = [normalize_type(T) for T in type(obj).__mro__]

    for scope in scope_stack:  # X-axis: context hierarchy
        for mro_type in mro:    # Y-axis: class hierarchy
            config = get_config_at_scope(scope, mro_type)
            if config and hasattr(config, field_name):
                value = getattr(config, field_name)
                if value is not None:
                    return (value, scope, mro_type)  # Provenance tuple
    return (None, None, None)
```

The algorithm walks two hierarchies simultaneously: scope_stack (global → plate → step) and MRO (child class → parent class). For each (scope, type) pair, it checks if a config of that type exists at that scope with a non-None value for the requested field.

The `mro_type` in the return tuple is the provenance: it records *which type* provided the value. This is only meaningful under nominal typing where `PathPlanningConfig` and `LazyPathPlanningConfig` are distinct despite identical structure. Duck typing sees both as having the same attributes, making `mro_type` meaningless.

MRO position encodes priority: types earlier in the MRO override later types. The dual-axis product (scope × MRO) creates O(|scopes| × |MRO|) checks in worst case, but terminates early on first match. Duck typing would require O(n) sequential attribute probing with no principled ordering.

Pattern (Table 5.1, Row 8): Dual-axis resolution with scope × MRO product. Demonstrates that provenance tracking fundamentally requires nominal identity (Corollary 6.3).

5.10 Case Study 9: Custom isinstance() Implementation

```
class GlobalConfigMeta(type):
    def __instancecheck__(cls, instance):
        # Virtual base class check
        if hasattr(instance.__class__, '_is_global_config'):
            return instance.__class__._is_global_config
        return super().__instancecheck__(instance)
```

```
# Usage: isinstance(config, GlobalConfigBase) returns True
# even if config doesn't inherit from GlobalConfigBase
```

This metaclass enables "virtual inheritance"—classes can satisfy `isinstance(obj, Base)` without explicitly inheriting from `Base`. The check relies on the `_is_global_config` class attribute (set by `@auto_create_decorator`), creating a nominal marker that duck typing cannot replicate.

Duck typing could check `hasattr(instance, '_is_global_config')`, but this is instance-level. The metaclass pattern requires class-level checks (`instance.__class__._is_global_config`), distinguishing the class from its instances. This is fundamentally nominal: the check is "does this type have this marker?" not "does this instance have this attribute?"

The virtual inheritance enables interface segregation: `GlobalPipelineConfig` advertises conformance to `GlobalConfigBase` without inheriting implementation. This is impossible with duck typing's attribute probing—there's no way to express "this class satisfies this interface" as a runtime-checkable property.

Pattern (Table 5.1, Row 9): Custom isinstance via class-level markers. Demonstrates that Python's metaobject protocol is fundamentally nominal.

5.11 Case Study 10: Dynamic Interface Generation

Pattern: Metaclass-generated abstract base classes create interfaces at runtime based on configuration. The generated ABCs have no methods or attributes—they exist purely for nominal identity.

```
class DynamicInterfaceMeta(ABCMeta):
    _generated_interfaces: Dict[str, Type] = {}

    @classmethod
    def get_or_create_interface(mcs, interface_name: str) -> Type:
        if interface_name not in mcs._generated_interfaces:
            # Generate pure nominal type
            interface = type(interface_name, (ABC,), {})
            mcs._generated_interfaces[interface_name] = interface
        return mcs._generated_interfaces[interface_name]

# Runtime usage
IStreamingConfig = DynamicInterfaceMeta.get_or_create_interface("IStreamingConfig")
class NapariConfig(StreamingConfig, IStreamingConfig): pass

# Later: isinstance(config, IStreamingConfig) $\rightarrow$ True
```

The generated interfaces have empty namespaces—no methods, no attributes. Their sole purpose is nominal identity: marking that a class explicitly claims to implement an interface. This is pure nominal typing: structural typing would see these interfaces as equivalent to `object` (since they have no distinguishing structure), but nominal typing distinguishes `IStreamingConfig` from `IVideoConfig` even though both are structurally empty.

Duck typing has no equivalent concept. There's no way to express "this class explicitly implements this contract" without actual attributes to probe. The nominal marker enables explicit interface

declarations in a dynamically-typed language.

Pattern (Table 5.1, Row 10): Runtime-generated interfaces with empty structure. Demonstrates that nominal identity can exist independent of structural content.

5.12 Case Study 11: Framework Detection via Sentinel Type

```
# Framework config uses sentinel type as registry key
_FRAMEWORK_CONFIG = type("_FrameworkConfigSentinel", (), {})()

# Detection: check if sentinel is registered
def has_framework_config():
    return _FRAMEWORK_CONFIG in GlobalRegistry.configs

# Alternative approaches fail:
# hasattr(module, '_FRAMEWORK_CONFIG') $\rightarrow$ fragile, module probing
# 'framework' in config_names $\rightarrow$ string-based, no type safety
```

The sentinel is a runtime-generated type with empty namespace, instantiated once, and used as a dictionary key. Its nominal identity (memory address) guarantees uniqueness—even if another module creates `type("_FrameworkConfigSentinel", (), {})()`, the two sentinels are distinct objects with distinct identities.

Duck typing cannot replicate this pattern. Attribute-based detection (`hasattr(module, attr_name)`) couples the check to module structure. String-based keys ('framework') lack type safety. The nominal sentinel provides a refactoring-safe, type-safe marker that exists independent of names or attributes.

This pattern appears in framework detection, feature flags, and capability markers—contexts where the existence of a capability needs to be checked without coupling to implementation details.

Pattern (Table 5.1, Row 11): Sentinel types for framework detection. Demonstrates nominal identity as a capability marker independent of structure.

5.13 Case Study 12: Dynamic Method Injection

```
def inject_conversion_methods(target_type: Type, methods: Dict[str, Callable]):
    """Inject methods into a type's namespace at runtime."""
    for method_name, method_impl in methods.items():
        setattr(target_type, method_name, method_impl)

# Usage: Inject GPU conversion methods into MemoryType converters
inject_conversion_methods(NumpyConverter, {
    'to_cupy': lambda self, data, gpu: cupy.asarray(data, gpu),
    'to_torch': lambda self, data, gpu: torch.tensor(data, device=gpu),
})
```

Method injection requires a target type—the type whose namespace will be modified. Duck typing has no concept of "the type itself" as a mutable namespace. It can only access instances. To inject methods duck-style would require modifying every instance's `__dict__`, which doesn't affect future instances.

The nominal type serves as a shared namespace. Injecting `to_cupy` into `NumpyConverter` affects all instances (current and future) because method lookup walks `type(obj).__dict__` before `obj.__dict__`. This is fundamentally nominal: the type is a first-class object with its own namespace, distinct from instance namespaces.

This pattern enables plugins, mixins, and monkey-patching—all requiring types as mutable namespaces. Duck typing's instance-level view cannot express "modify the behavior of all objects of this kind."

Pattern (Table 5.1, Row 12): Dynamic method injection into type namespaces. Demonstrates that Python's type system treats types as first-class objects with nominal identity.

5.14 Case Study 13: Bidirectional Type Lookup

OpenHCS maintains bidirectional registries linking lazy types to base types: `_lazy_to_base[LazyX] = X` and `_base_to_lazy[X] = LazyX`. How should the system prevent desynchronization bugs where the two dicts fall out of sync?

```
class BidirectionalTypeRegistry:
    def __init__(self):
        self._forward: Dict[Type, Type] = {}  # lazy $\rightarrow$ base
        self._reverse: Dict[Type, Type] = {}  # base $\rightarrow$ lazy

    def register(self, lazy_type: Type, base_type: Type):
        # Single source of truth: type identity enforces bijection
        if lazy_type in self._forward:
            raise ValueError(f"{lazy_type} already registered")
        if base_type in self._reverse:
            raise ValueError(f"{base_type} already has lazy companion")

        self._forward[lazy_type] = base_type
        self._reverse[base_type] = lazy_type

# Type identity as key ensures sync
registry.register(LazyPathPlanningConfig, PathPlanningConfig)
# Later: registry.normalize(LazyPathPlanningConfig) $\rightarrow$ PathPlanningConfig
#        registry.get_lazy(PathPlanningConfig) $\rightarrow$ LazyPathPlanningConfig
```

Duck typing would require maintaining two separate dicts with string keys (class names), introducing synchronization bugs. Renaming `PathPlanningConfig` would break the string-based lookup. The nominal type identity serves as a refactoring-safe key that guarantees both dicts stay synchronized—a type can only be registered once, enforcing bijection.

The registry operations are O(1) lookups by type identity. Duck typing's string-based approach would require O(n) string matching or maintaining parallel indices, both error-prone and slower.

Pattern (Table 5.1, Row 13): Bidirectional type registries with synchronization guarantees. Demonstrates that nominal identity as dict key prevents desynchronization bugs inherent to string-based approaches.

---

6. Formalization and Verification

We provide machine-checked proofs of our core theorems in Lean 4. The complete development (758 lines, 0 `sorry` placeholders) includes the abstract class system model, axis lattice metatheorem, nominal resolution algorithm, and a formalization of duck typing's impossibility for provenance tracking.

6.1 Type Universe and Registry

Types are represented as natural numbers, capturing nominal identity:

```
-- Types are represented as natural numbers (nominal identity)
abbrev Typ := Nat

-- The lazy-to-base registry as a partial function
def Registry := Typ $\rightarrow$ Option Typ

-- A registry is well-formed if base types are not in domain
def Registry.wellFormed (R : Registry) : Prop :=
  $\forall$ L B, R L = some B $\rightarrow$ R B = none

-- Normalization: map lazy type to base, or return unchanged
def normalizeType (R : Registry) (T : Typ) : Typ :=
  match R T with
  | some B => B
  | none => T
```

Invariant (Normalization Idempotence). For well-formed registries, normalization is idempotent:

```
theorem normalizeType_idempotent (R : Registry) (T : Typ)
    (h_wf : R.wellFormed) :
    normalizeType R (normalizeType R T) = normalizeType R T := by
  simp only [normalizeType]
  cases hR : R T with
  | none => simp only [hR]
  | some B =>
    have h_base : R B = none := h_wf T B hR
    simp only [h_base]
```

6.2 MRO and Scope Stack

```
-- MRO is a list of types, most specific first
abbrev MRO := List Typ

-- Scope stack: most specific first
abbrev ScopeStack := List ScopeId

-- Config instance: type and field value
structure ConfigInstance where
```

```
  typ : Typ
  fieldValue : FieldValue

-- Configs available at each scope
def ConfigContext := ScopeId $\rightarrow$ List ConfigInstance
```

6.3 The RESOLVE Algorithm

```
-- Resolution result: value, scope, source type
structure ResolveResult where
  value : FieldValue
  scope : ScopeId
  sourceType : Typ
deriving DecidableEq

-- Find first matching config in a list
def findConfigByType (configs : List ConfigInstance) (T : Typ) :
    Option FieldValue :=
  match configs.find? (fun c => c.typ == T) with
  | some c => some c.fieldValue
  | none => none

-- The dual-axis resolution algorithm
def resolve (R : Registry) (mro : MRO)
    (scopes : ScopeStack) (ctx : ConfigContext) :
    Option ResolveResult :=
  -- X-axis: iterate scopes (most to least specific)
  scopes.findSome? fun scope =>
    -- Y-axis: iterate MRO (most to least specific)
    mro.findSome? fun mroType =>
      let normType := normalizeType R mroType
      match findConfigByType (ctx scope) normType with
      | some v =>
        if v $\neq$ 0 then some ⟨v, scope, normType⟩
        else none
      | none => none
```

6.4 GETATTRIBUTE Implementation

```
-- Raw field access (before resolution)
def rawFieldValue (obj : ConfigInstance) : FieldValue :=
  obj.fieldValue

-- GETATTRIBUTE implementation
def getattribute (R : Registry) (obj : ConfigInstance) (mro : MRO)
    (scopes : ScopeStack) (ctx : ConfigContext) (isLazyField : Bool) :
    FieldValue :=
  let raw := rawFieldValue obj
```

```
  if raw $\neq$ 0 then raw  -- Concrete value, no resolution
  else if isLazyField then
    match resolve R mro scopes ctx with
    | some result => result.value
    | none => 0
  else raw
```

6.5 Theorem 6.1: Resolution Completeness

Theorem 6.1 (Completeness). The `resolve` function is complete: it returns value `v` if and only if either no resolution occurred (v = 0) or a valid resolution result exists.

```
theorem resolution_completeness
    (R : Registry) (mro : MRO)
    (scopes : ScopeStack) (ctx : ConfigContext) (v : FieldValue) :
    (match resolve R mro scopes ctx with
     | some r => r.value
     | none => 0) = v $\leftrightarrow$
    (v = 0 $\land$ resolve R mro scopes ctx = none) $\lor$
    ($\exists$ r : ResolveResult,
      resolve R mro scopes ctx = some r $\land$ r.value = v) := by
  cases hr : resolve R mro scopes ctx with
  | none =>
    constructor
    · intro h; left; exact ⟨h.symm, rfl⟩
    · intro h
      rcases h with ⟨hv, _⟩ | ⟨r, hfalse, _⟩
      · exact hv.symm
      · cases hfalse
  | some result =>
    constructor
    · intro h; right; exact ⟨result, rfl, h⟩
    · intro h
      rcases h with ⟨_, hfalse⟩ | ⟨r, hr2, hv⟩
      · cases hfalse
      · simp only [Option.some.injEq] at hr2
        rw [$\leftarrow$ hr2] at hv; exact hv
```

6.6 Theorem 6.2: Provenance Preservation

Theorem 6.2a (Uniqueness). Resolution is deterministic: same inputs always produce the same result.

```
theorem provenance_uniqueness
    (R : Registry) (mro : MRO) (scopes : ScopeStack) (ctx : ConfigContext)
    (result_1 result_2 : ResolveResult)
    (hr_1 : resolve R mro scopes ctx = some result_1)
    (hr_2 : resolve R mro scopes ctx = some result_2) :
    result_1 = result_2 := by
```

```
  simp only [hr_1, Option.some.injEq] at hr_2
  exact hr_2
```

Theorem 6.2b (Determinism). Resolution function is deterministic.

```
theorem resolution_determinism
    (R : Registry) (mro : MRO) (scopes : ScopeStack) (ctx : ConfigContext) :
    $\forall$ r_1 r_2, resolve R mro scopes ctx = r_1 $\rightarrow$
             resolve R mro scopes ctx = r_2 $\rightarrow$
             r_1 = r_2 := by
  intros r_1 r_2 h_1 h_2
  rw [$\leftarrow$ h_1, $\leftarrow$ h_2]
```

6.7 Duck Typing Formalization

We now formalize duck typing and prove it cannot provide provenance.

Duck object structure:

```
-- In duck typing, a "type" is just a bag of (field_name, field_value) pairs
-- There's no nominal identity - only structure matters
structure DuckObject where
  fields : List (String $\times$ Nat)
deriving DecidableEq

-- Field lookup in a duck object
def getField (obj : DuckObject) (name : String) : Option Nat :=
  match obj.fields.find? (fun p => p.1 == name) with
  | some p => some p.2
  | none => none
```

Structural equivalence:

```
-- Two duck objects are "structurally equivalent" if they have same fields
-- This is THE defining property of duck typing: identity = structure
def structurallyEquivalent (a b : DuckObject) : Prop :=
  $\forall$ name, getField a name = getField b name
```

We prove this is an equivalence relation:

```
theorem structEq_refl (a : DuckObject) :
  structurallyEquivalent a a := by
  intro name; rfl

theorem structEq_symm (a b : DuckObject) :
    structurallyEquivalent a b $\rightarrow$ structurallyEquivalent b a := by
  intro h name; exact (h name).symm

theorem structEq_trans (a b c : DuckObject) :
    structurallyEquivalent a b $\rightarrow$ structurallyEquivalent b c $\rightarrow$
    structurallyEquivalent a c := by
  intro hab hbc name; rw [hab name, hbc name]
```

The Duck Typing Axiom:

Any function operating on duck objects must respect structural equivalence. If two objects have the same structure, they are indistinguishable. This is not an assumption—it is the *definition* of duck typing: "If it walks like a duck and quacks like a duck, it IS a duck."

```
-- A duck-respecting function treats structurally equivalent objects identically
def DuckRespecting (f : DuckObject $\rightarrow$ $\alpha$) : Prop :=
  $\forall$ a b, structurallyEquivalent a b $\rightarrow$ f a = f b
```

6.8 Corollary 6.3: Duck Typing Cannot Provide Provenance

Provenance requires returning WHICH object provided a value. But in duck typing, structurally equivalent objects are indistinguishable. Therefore, any "provenance" must be constant on equivalent objects.

```
-- Suppose we try to build a provenance function for duck typing
-- It would have to return which DuckObject provided the value
structure DuckProvenance where
  value : Nat
  source : DuckObject  -- "Which object provided this?"
deriving DecidableEq
```

Theorem (Indistinguishability). Any duck-respecting provenance function cannot distinguish sources:

```
theorem duck_provenance_indistinguishable
    (getProvenance : DuckObject $\rightarrow$ Option DuckProvenance)
    (h_duck : DuckRespecting getProvenance)
    (obj1 obj2 : DuckObject)
    (h_equiv : structurallyEquivalent obj1 obj2) :
    getProvenance obj1 = getProvenance obj2 := by
  exact h_duck obj1 obj2 h_equiv
```

Corollary 6.3 (Absurdity). If two objects are structurally equivalent and both provide provenance, the provenance must claim the SAME source for both (absurd if they're different objects):

```
theorem duck_provenance_absurdity
    (getProvenance : DuckObject $\rightarrow$ Option DuckProvenance)
    (h_duck : DuckRespecting getProvenance)
    (obj1 obj2 : DuckObject)
    (h_equiv : structurallyEquivalent obj1 obj2)
    (prov1 prov2 : DuckProvenance)
    (h1 : getProvenance obj1 = some prov1)
    (h2 : getProvenance obj2 = some prov2) :
    prov1 = prov2 := by
  have h_eq := h_duck obj1 obj2 h_equiv
  rw [h1, h2] at h_eq
  exact Option.some.inj h_eq
```

The key insight: In duck typing, if `obj1` and `obj2` have the same fields, they are structurally equivalent. Any duck-respecting function returns the same result for both. Therefore, provenance

CANNOT distinguish them. Therefore, provenance is IMPOSSIBLE in duck typing.

Contrast with nominal typing: In our nominal system, types are distinguished by identity:

```
-- Example: Two nominally different types
def WellFilterConfigType : Nat := 1
def StepWellFilterConfigType : Nat := 2

-- These are distinguishable despite potentially having same structure
theorem nominal_types_distinguishable :
    WellFilterConfigType $\neq$ StepWellFilterConfigType := by decide
```

Therefore, `ResolveResult.sourceType` is meaningful: it tells you WHICH type provided the value, even if types have the same structure.

6.9 Verification Status

Component	Lines	Status
AbstractClassSystem namespace	475	PASS Compiles, no warnings
- Three-axis model (N, B, S)	80	PASS Definitions
- Typing discipline capabilities	100	PASS Proved
- Strict dominance (Theorem 2.11)	60	PASS Proved
- Mixin dominance (Theorem 8.1)	80	PASS Proved
- Axis lattice metatheorem	90	PASS Proved
- Information-theoretic completeness	65	PASS Proved
NominalResolution namespace	157	PASS Compiles, no warnings
- Type definitions & registry	40	PASS Proved
- Normalization idempotence	12	PASS Proved
- MRO & scope structures	30	PASS Compiles
- RESOLVE algorithm	25	PASS Compiles
- Theorem 6.1 (completeness)	25	PASS Proved
- Theorem 6.2 (uniqueness)	25	PASS Proved
DuckTyping namespace	127	PASS Compiles, no warnings
- DuckObject structure	20	PASS Compiles
- Structural equivalence	30	PASS Proved (equivalence relation)
- Duck typing axiom	10	PASS Definition
- Corollary 6.3 (impossibility)	40	PASS Proved
- Nominal contrast	10	PASS Proved
Total	**758**	**PASS All proofs verified, 0 `sorry`**

6.10 What the Lean Proofs Guarantee

The machine-checked verification establishes:

1. **Algorithm correctness**: `resolve` returns value `v` iff resolution found a config providing `v` (Theorem 6.1).
2. **Determinism**: Same inputs always produce same `(value, scope, sourceType)` tuple (Theorem 6.2).

3. **Idempotence**: Normalizing an already-normalized type is a no-op (normalization_idempotent).

4. **Duck typing impossibility**: Any function respecting structural equivalence cannot distinguish between structurally identical objects, making provenance tracking impossible (Corollary 6.3).

What the proofs do NOT guarantee:

- **C3 correctness**: We assume MRO is well-formed. Python's C3 algorithm can fail on pathological diamonds (raising `TypeError`). Our proofs apply only when C3 succeeds.
- **Registry invariants**: `Registry.wellFormed` is an axiom (base types not in domain). We prove theorems *given* this axiom but do not derive it from more primitive foundations.
- **Termination**: We use Lean's termination checker to verify `resolve` terminates, but the complexity bound O(|scopes| × |MRO|) is informal, not mechanically verified.

This is standard practice in mechanized verification: CompCert assumes well-typed input, seL4 assumes hardware correctness. Our proofs establish that *given* a well-formed registry and MRO, the resolution algorithm is correct and provides provenance that duck typing cannot.

6.11 External Provenance Map Rebuttal

Objection: "Duck typing could provide provenance via an external map: `provenance_map: Dict[id(obj), SourceType]`."

Rebuttal: This objection conflates *object identity* with *type identity*. The external map tracks which specific object instance came from where—not which *type* in the MRO provided a value.

Consider:

```
class A:
    x = 1

class B(A):
    pass  # Inherits x from A

b = B()
print(b.x)  # Prints 1. Which type provided this?
```

An external provenance map could record `provenance_map[id(b)] = B`. But this doesn't answer the question "which type in B's MRO provided `x`?" The answer is `A`, and this requires MRO traversal—which requires the Bases axis.

Formal statement: Let ExternalMap : ObjectId → SourceType be any external provenance map. Then:

ExternalMap cannot answer: "Which type in MRO(type(obj)) provided attribute a?"

Proof. The question asks about MRO position. MRO is derived from Bases. ExternalMap has no access to Bases (it maps object IDs to types, not types to MRO positions). Therefore ExternalMap cannot answer MRO-position queries. ■

The deeper point: Provenance is not about "where did this object come from?" It's about "where did this *value* come from in the inheritance hierarchy?" The latter requires MRO, which requires Bases, which duck typing discards.

6.12 Abstract Model Lean Formalization

The abstract class system model (Section 2.4) is formalized in Lean 4 with complete proofs (no `sorry` placeholders):

```
-- The three axes of a class system
inductive Axis where
  | Name       -- N: type identifier
  | Bases      -- B: inheritance hierarchy
  | Namespace  -- S: attribute declarations (shape)
deriving DecidableEq, Repr

-- A typing discipline is characterized by which axes it inspects
abbrev AxisSet := List Axis

-- Canonical axis sets
def shapeAxes : AxisSet := [.Name, .Namespace]  -- Structural/duck typing
def nominalAxes : AxisSet := [.Name, .Bases, .Namespace]  -- Full nominal

-- Unified capability (combines typing and architecture domains)
inductive UnifiedCapability where
  | interfaceCheck      -- Check interface satisfaction
  | identity            -- Type identity
  | provenance          -- Type provenance
  | enumeration         -- Subtype enumeration
  | conflictResolution  -- MRO-based resolution
deriving DecidableEq, Repr

-- Capabilities enabled by each axis
def axisCapabilities (a : Axis) : List UnifiedCapability :=
  match a with
  | .Name => [.interfaceCheck]
  | .Bases => [.identity, .provenance, .enumeration, .conflictResolution]
  | .Namespace => [.interfaceCheck]

-- Capabilities of an axis set = union of each axis's capabilities
def axisSetCapabilities (axes : AxisSet) : List UnifiedCapability :=
  axes.flatMap axisCapabilities |>.eraseDups
```

Theorem 6.4 (Axis Lattice — Lean). Shape capabilities are a strict subset of nominal capabilities:

```
-- THEOREM: Shape axes $\subset$ Nominal axes (specific instance of lattice ordering)
theorem axis_shape_subset_nominal :
    $\forall$ c $\in$ axisSetCapabilities shapeAxes,
```

```
      c $\in$ axisSetCapabilities nominalAxes := by
  intro c hc
  have h_shape : axisSetCapabilities shapeAxes = [UnifiedCapability.interfaceCheck] := rfl
  have h_nominal : UnifiedCapability.interfaceCheck $\in$ axisSetCapabilities nominalAxes := by
  rw [h_shape] at hc
  simp only [List.mem_singleton] at hc
  rw [hc]
  exact h_nominal

-- THEOREM: Nominal has capabilities Shape lacks
theorem axis_nominal_exceeds_shape :
    $\exists$ c $\in$ axisSetCapabilities nominalAxes,
      c $\notin$ axisSetCapabilities shapeAxes := by
  use UnifiedCapability.provenance
  constructor
  · decide  -- provenance $\in$ nominalAxes capabilities
  · decide  -- provenance $\notin$ shapeAxes capabilities

-- THE LATTICE METATHEOREM: Combined strict dominance
theorem lattice_dominance :
    ($\forall$ c $\in$ axisSetCapabilities shapeAxes, c $\in$ axisSetCapabilities nominalAxes)
    ($\exists$ c $\in$ axisSetCapabilities nominalAxes, c $\notin$ axisSetCapabilities shapeAx
  ⟨axis_shape_subset_nominal, axis_nominal_exceeds_shape⟩
```

This formalizes Theorem 2.9: using more axes provides strictly more capabilities. The proofs are complete and compile without any `sorry` placeholders.

---

7. Related Work

7.1 Type Theory Foundations

Malayeri & Aldrich (ECOOP 2008, ESOP 2009). The foundational work on integrating nominal and structural subtyping. Their ECOOP 2008 paper "Integrating Nominal and Structural Subtyping" proves type safety for a combined system, but explicitly states that neither paradigm is strictly superior. They articulate the key distinction: *"Nominal subtyping lets programmers express design intent explicitly (checked documentation of how components fit together)"* while *"structural subtyping is far superior in contexts where the structure of the data is of primary importance."* Critically, they observe that structural typing excels at **retrofitting** (integrating independently-developed components), whereas nominal typing aligns with **planned, integrated designs**. Their ESOP 2009 empirical study found that adding structural typing to Java would benefit many codebases—but they also note *"there are situations where nominal types are more appropriate"* and that without structural typing, interface proliferation would explode by ~300%.

Our contribution: We extend their qualitative observation into a formal claim: in *greenfield* systems with explicit inheritance hierarchies (like OpenHCS), nominal typing is not just "appropriate" but *necessary* for capabilities like provenance tracking and MRO-based resolution.

Abdelgawad & Cartwright (ENTCS 2014). Their domain-theoretic model NOOP proves that

in nominal languages, **inheritance and subtyping become identical**—formally validating the intuition that declaring a subclass makes it a subtype. They contrast this with Cook et al. (1990)'s structural claim that "inheritance is not subtyping," showing that the structural view ignores nominal identity. Key insight: purely structural OO typing admits **spurious subtyping**—a type can accidentally be a subtype due to shape alone, violating intended contracts.

Our contribution: OpenHCS's dual-axis resolver depends on this identity. The resolution algorithm walks `type(obj).__mro__` precisely because MRO encodes the inheritance hierarchy as a total order. If subtyping and inheritance could diverge (as in structural systems), the algorithm would be unsound.

Abdelgawad (arXiv 2016). The essay "Why Nominal-Typing Matters in OOP" argues that nominal typing provides **information centralization**: *"objects and their types carry class names information as part of their meaning"* and those names correspond to behavioral contracts. Type names aren't just shapes—they imply specific intended semantics. Structural typing, treating objects as mere records, *"cannot naturally convey such semantic intent."*

Our contribution: Theorem 6.2 (Provenance Preservation) formalizes this intuition. The tuple `(value, scope_id, source_type)` returned by `resolve` captures exactly the "class name information" that Abdelgawad argues is essential. Duck typing loses this information after attribute access.

7.2 Practical Hybrid Systems

Gil & Maman (OOPSLA 2008). Whiteoak adds structural typing to Java for **retrofitting**—treating classes as subtypes of structural interfaces without modifying source. Their motivation: *"many times multiple classes have no common supertype even though they could share an interface."* This supports the Malayeri-Aldrich observation that structural typing's benefits are context-dependent.

Our contribution: OpenHCS is explicitly **greenfield**—the entire config framework was designed with nominal typing from the start. The capabilities demonstrated (MRO-based resolution, bidirectional type registries, provenance tracking) would be impossible to retrofit into a structural system.

Go (2012) and TypeScript (2012+). Both adopt structural typing for pragmatic reasons: - Go uses structural interface satisfaction to reduce boilerplate. - TypeScript uses structural compatibility to integrate with JavaScript's untyped ecosystem.

However, both face the **accidental compatibility problem**. TypeScript developers use "branding" (adding nominal tag properties) to differentiate structurally identical types—a workaround that **reintroduces nominal typing**. The TypeScript issue tracker has open requests for native nominal types.

Our contribution: OpenHCS avoids this problem by using nominal typing from the start. The `@global_pipeline_config` chain generates `LazyPathPlanningConfig` as a distinct type from `PathPlanningConfig` precisely to enable different behavior (resolution on access) while sharing the same structure.

7.3 Metaprogramming Complexity

Veldhuizen (2006). "Tradeoffs in Metaprogramming" proves that sufficiently expressive metaprogramming can yield **unbounded savings** in code length—Blum (1967) showed that restricting a powerful language causes non-computable blow-up in program size. This formally underpins our use of `make_dataclass()` to generate companion types.

Proposition: Multi-stage metaprogramming is no more powerful than one-stage generation for the class of computable functions.

Our contribution: The 5-stage `@global_pipeline_config` chain is not nested metaprogramming (programs generating programs generating programs)—it's a single-stage generation that happens to have 5 sequential phases. This aligns with Veldhuizen's bound: we achieve full power without complexity explosion.

Damaševičius & Štuikys (2010). They define metrics for metaprogram complexity: - **Relative Kolmogorov Complexity (RKC):** compressed/actual size - **Cognitive Difficulty (CD):** chunks of meta-information to hold simultaneously

They found that C++ Boost template metaprogramming can be "over-complex" when abstraction goes too far.

Our contribution: OpenHCS's metaprogramming is **homogeneous** (Python generating Python) rather than heterogeneous (separate code generators). Their research shows homogeneous metaprograms have lower complexity overhead. Our decorators read as declarative annotations, not as complex template metaprograms.

7.4 Behavioral Subtyping

Liskov & Wing (1994). The Liskov Substitution Principle formally defines behavioral subtyping: *"any property proved about supertype objects should hold for its subtype objects."* Nominal typing enables this by requiring explicit `is-a` declarations.

Our contribution: The `@global_pipeline_config` chain enforces behavioral subtyping through field inheritance with modified defaults. When `LazyPathPlanningConfig` inherits from `PathPlanningConfig`, it **must** have the same fields (guaranteed by runtime type generation), but with `None` defaults (different behavior). The nominal type system tracks that these are distinct types with different resolution semantics.

7.5 Positioning This Work

Work	Contribution	Our Extension
Malayeri & Aldrich	Qualitative trade-offs	Formal necessity claim for greenfield
Abdelgawad & Cartwright	Inheritance = subtyping in nominal	MRO-based resolution algorithm
Abdelgawad	Information centralization	Provenance as formal tuple
Veldhuizen	Metaprogramming bounds	5-stage chain respects bounds
Liskov & Wing	Behavioral subtyping	Field inheritance enforcement

Our core contribution: Prior work established that nominal and structural typing have trade-offs. We prove that for systems requiring **provenance tracking** (which type provided a value), **MRO-based resolution** (inheritance hierarchy determines priority), and **bidirectional type registries** (LazyX $\leftrightarrow$ X mapping), nominal typing is not just preferable but **necessary**. Duck typing is proven strictly dominated: it cannot provide these capabilities at any cost.

---

8. Discussion

8.1 Limitations

Our theorems establish necessary conditions for provenance-tracking systems, but several limitations warrant explicit acknowledgment:

Diamond inheritance. Our theorems assume well-formed MRO produced by C3 linearization. Pathological diamond inheritance patterns can break C3 entirely—Python raises `TypeError` when linearization fails. Such cases require manual resolution or interface redesign. Our complexity bounds apply only when C3 succeeds.

Runtime overhead. Provenance tracking stores `(value, scope_id, source_type)` tuples for each resolved field. This introduces memory overhead proportional to the number of lazy fields. In OpenHCS, this overhead is negligible ($< 1\%$ of total memory usage), but systems with millions of configuration objects may need to consider this cost.

Not universal. Simple scripts, one-off data analysis tools, and prototype code do not benefit from provenance tracking. Duck typing remains appropriate for small programs where error localization is trivial (the entire program fits in working memory). Our impossibility theorem applies only when provenance is a requirement.

Python-specific foundations. Our theorems rely on Python's specific implementation of `type(name, bases, namespace)` and C3 linearization. While the conceptual results (nominal typing for provenance, $O(1)$ vs $\Omega(n)$ complexity) generalize to other nominal languages, the precise formalization is Python-specific. Section 8.4 discusses implications for other languages.

Metaclass complexity. The `@global_pipeline_config` chain (Case Study 7) requires understanding five metaprogramming stages: decorator invocation, metaclass `__prepare__`, descriptor `__set_name__`, field injection, and type registration. This complexity is manageable in OpenHCS because it's encapsulated in a single decorator, but unconstrained metaclass composition can lead to maintenance challenges.

Lean proofs assume well-formedness. Our Lean 4 verification includes `Registry.wellFormed` and MRO monotonicity as axioms rather than derived properties. We prove theorems *given* these axioms, but do not prove the axioms themselves from more primitive foundations. This is standard practice in mechanized verification (e.g., CompCert assumes well-typed input), but limits the scope of our machine-checked guarantees.

8.2 When Shape-Based Typing Is a Valid Concession

Theorem 3.1 establishes that structural typing is valid for "namespace-only" classes—those lacking explicit inheritance. This explains when shape-based typing is an acceptable concession:

Retrofit scenarios. When integrating independently developed components that share no common base classes, you cannot mandate inheritance from your base classes. Structural typing is the only option. This is a concession to code you do not control—not a design choice.

Languages without inheritance. Go's struct types have no inheritance axis (`bases = []`), so structural typing is both necessary and sufficient. Our Corollary 3.2 formalizes this: structural typing is correct when `bases = []` universally. This is why Go was designed this way—not because structural typing is superior, but because Go lacks inheritance.

Library boundaries. At module boundaries where explicit inheritance is unavailable, structural constraints are the only option. Theorem 3.1 applies: the constraint is structural because there is no shared `bases` to use.

To be clear: in these contexts, shape-based typing is an acceptable concession. It is never the correct choice when you control the type hierarchy. Our contribution is proving that shape-based typing is categorically wrong for greenfield systems with provenance requirements—not merely suboptimal, but incapable of providing the required properties.

8.3 Future Work

Extension to other nominal languages. Java, C++, Scala, and Rust all couple nominal typing with inheritance, but their type construction mechanisms differ from Python's `type()`. Formalizing the general principle—provenance requires nominal identity—in a language-agnostic framework remains open.

Formalization of greenfield-retrofit distinction. We currently define "greenfield" as "programmer can choose `bases`" and "retrofit" as "no shared `bases` available." A more rigorous treatment would formalize when each regime applies and prove decidability of regime classification.

Gradual nominal/structural typing. TypeScript supports both nominal (via branding) and structural typing in the same program. Formalizing the interaction between these disciplines, and proving soundness of gradual migration, would enable principled adoption strategies.

Trait systems and mixins. Rust traits and Scala mixins provide multiple inheritance of behavior without nominal base classes. Our theorems apply to Python's MRO, but trait resolution uses different algorithms. Extending our complexity bounds to trait systems would broaden applicability.

Automated complexity inference. Given a type system specification, can we automatically compute whether error localization is $O(1)$ or $\Omega(n)$? Such a tool would help language designers evaluate typing discipline tradeoffs during language design.

8.4 Implications for Language Design

Language designers face a fundamental choice: provide nominal typing (enabling provenance), structural typing (enabling retrofit), or both. Our theorems inform this decision:

Provide both mechanisms. Languages like TypeScript demonstrate that nominal and structural typing can coexist. TypeScript's "branding" idiom (using private fields to create nominal distinctions) validates our thesis: programmers need nominal identity even in structurally-typed languages. Python provides both ABCs (nominal) and `Protocol` (structural). Our theorems clarify when each is correct: `Protocol` is for retrofit boundaries where you cannot mandate inheritance; ABCs are for

greenfield code where you control the hierarchy. Using `Protocol` in greenfield code is wrong—it discards the inheritance information you control.

MRO-based resolution is near-optimal. Python's descriptor protocol combined with C3 linearization achieves O(1) field resolution while preserving provenance. Languages designing new metaobject protocols should consider whether they can match this complexity bound.

Explicit `bases` mandates nominal typing. If a language exposes explicit inheritance declarations (`class C(Base)`), Theorem 3.4 applies: structural typing becomes insufficient. Language designers cannot add inheritance to a structurally-typed language without addressing the provenance requirement.

8.5 Derivable Code Quality Metrics

The formal model yields four measurable metrics that can be computed statically from source code:

Metric 1: Duck Typing Density (DTD)

```
DTD = (hasattr_calls + getattr_calls + try_except_attributeerror) / KLOC
```

Measures ad-hoc runtime probing. High DTD in greenfield code indicates discipline violation. High DTD at module boundaries (retrofit) is acceptable.

Metric 2: Nominal Typing Ratio (NTR)

```
NTR = (isinstance_calls + type_as_dict_key + abc_registrations) / KLOC
```

Measures explicit type contracts. High NTR indicates intentional use of inheritance hierarchy.

Metric 3: Provenance Capability (PC) Binary metric: does the codebase contain queries of the form "which type provided this value"? Presence of `(value, scope, source_type)` tuples, MRO traversal for resolution, or `type(obj).__mro__` inspection indicates PC = 1. If PC = 1, nominal typing is mandatory (Corollary 6.3).

Metric 4: Resolution Determinism (RD)

```
RD = mro_based_dispatch / (mro_based_dispatch + runtime_probing_dispatch)
```

Measures O(1) vs $\Omega(n)$ error localization. RD = 1 indicates all dispatch is MRO-based (nominal). RD = 0 indicates all dispatch is runtime probing (duck).

Tool implications: These metrics enable automated linters. A linter could flag `hasattr()` in greenfield modules (DTD violation), suggest `isinstance()` replacements, and verify that provenance-tracking codebases maintain NTR above a threshold.

Empirical application: In OpenHCS, DTD dropped from 47 calls in the UI layer (before PR #44) to 0 after migration. NTR increased correspondingly. PC = 1 throughout (dual-axis resolver requires provenance). RD = 1 (all dispatch is MRO-based).

8.6 Hybrid Systems and Methodology Scope

Our theorems establish necessary conditions for provenance-tracking systems. This section clarifies when the methodology applies and when shape-based typing is an acceptable concession.

8.6.1 When Shape-Based Typing Is Acceptable **Retrofit scenarios.** When integrating independently developed components that share no common base classes, you cannot mandate inheritance from your base classes. Structural typing is the only option. This is a concession to code you do not control—not a design choice.

Language boundaries. Calling from Python into C libraries, where inheritance relationships are unavailable. The C struct has no `bases` axis, making structural checking the only option.

Versioning and compatibility. When newer code must accept older types that predate a base class introduction. Example: A library adds `ConfigBase` in v2.0 but must accept v1.0 configs lacking that base.

Type-level programming without runtime overhead. TypeScript's structural typing enables type checking at compile time without runtime cost. For TypeScript code that never uses `instanceof` or class identity, structural typing is an acceptable design. However, see Section 8.7 for why TypeScript's *class-based* structural typing is problematic.

8.6.2 The Greenfield Criterion A system is "greenfield" with respect to a type hierarchy if: 1. The architect can modify type definitions to add/remove base classes 2. All implementing types are within the system's codebase (not external) 3. There is no requirement to accept "foreign" types from untrusted sources

Example: OpenHCS's configuration system is greenfield because all config types are defined in the project codebase. The architect can mandate `class PathPlanningConfig(GlobalConfigBase)` and enforce this throughout.

Counter-example: A JSON schema validator is not greenfield with respect to JSON objects because it must accept externally-defined JSON from API responses. Structural validation ("does this JSON have the required fields?") is the only option.

8.6.3 Hybrid Boundaries Systems often have both greenfield and retrofit components. The methodology applies per-component:

```python
# Greenfield: internal config hierarchy (use nominal)
class ConfigBase(ABC):
    @abstractmethod
    def validate(self) -> bool: pass

class PathPlanningConfig(ConfigBase):
    well_filter: Optional[str]

# Retrofit: accept external dicts (use structural)
def load_config_from_json(json_dict: Dict[str, Any]) -> ConfigBase:
    # Structural check: does JSON have required fields?
    if "well_filter" in json_dict:
        return PathPlanningConfig(**json_dict)
    raise ValueError("Invalid config")
```

The greenfield component (`ConfigBase` hierarchy) uses nominal typing. The retrofit boundary (`load_config_from_json`) uses structural validation because external JSON has no inheritance. This is correct: use nominal where you control types, structural at boundaries where you don't.

8.6.4 Scope Summary

Context	Typing Discipline	Justification
Greenfield + provenance required	Nominal (mandatory)	Theorem 3.5, Corollary 6.3
Greenfield + no provenance	Nominal (recommended)	Theorem 3.5 (strict dominance)
Retrofit / external types	Structural (acceptable)	Theorem 3.1 (namespace-only)
Small scripts / prototypes	Duck (acceptable)	Complexity cost is negligible
Language boundaries (C/FFI)	Structural (mandatory)	No inheritance available

The methodology does not claim "always use nominal typing." It claims "in greenfield development, nominal typing is correct; shape-based typing is a concession to constraints, not a design choice."

8.7 Case Study: TypeScript's Design Incoherence

TypeScript presents a puzzle: it has explicit inheritance (`class B extends A`) but uses structural subtyping. Is this a valid design tradeoff, or a mistake?

8.7.1 The Incoherence Argument TypeScript's design exhibits three forms of incoherence:

Incoherence 1: Adding then ignoring semantic information.

```
class WellFilterConfig {
    well_filter: string | null;
}

class StepWellFilterConfig extends WellFilterConfig {
    // Structurally identical to parent
}

// TypeScript treats these as equivalent types
function f(config: WellFilterConfig) { }
f(new StepWellFilterConfig());  // OK - structural compatibility
f(new WellFilterConfig());      // OK - same structure

// But the programmer declared a distinction via 'extends'
// TypeScript ignores this declaration for type checking
```

The programmer wrote `extends` to indicate a semantic relationship. TypeScript's type checker discards this information. This is incoherence: the language provides a mechanism (`extends`) then ignores it.

Incoherence 2: Branding as a workaround.

TypeScript programmers use "branding" to create nominal distinctions:

```
// Workaround: add a private field to force nominal distinction
class StepWellFilterConfig extends WellFilterConfig {
    private __brand!: void;  // Forces nominal identity
}

// Now TypeScript treats them as distinct (private field differs)
```

This is an admission that structural typing is insufficient. If structural typing were correct, branding would be unnecessary.

Incoherence 3: Runtime vs compile-time mismatch.

```
class A { x: number = 1; }
class B { x: number = 1; }

// Compile time: structural equivalence
function f(a: A) { }
f(new B());  // OK - same structure

// Runtime: instanceof checks nominal identity
const b = new B();
console.log(b instanceof A);  // false - different classes
```

TypeScript's type system says `A` and `B` are equivalent, but JavaScript's runtime says they're distinct. This creates a semantic gap: code that type-checks may fail at runtime if it uses `instanceof`.

8.7.2 Connecting to Our Theorems Our Theorem 3.4 states: "The presence of a `bases` axis mandates nominal typing." TypeScript violates this:

- TypeScript has `bases` (via `extends`)
- TypeScript uses structural typing (ignoring `bases`)
- Therefore TypeScript exhibits design incoherence per Theorem 3.4

Our Corollary 3.6 states: "Choosing structural typing in greenfield is incorrect." TypeScript's class syntax is greenfield (the programmer defines the classes), so structural typing is incorrect for classes per our theorems.

8.7.3 Implications for Language Design TypeScript demonstrates that adding `class` syntax to a language creates an expectation of nominal typing. When the type system violates this expectation (via structural checking), users create workarounds (branding) to restore nominal behavior.

The lesson: If your language has explicit inheritance (`class B extends A`), your type system should respect it. Structural typing is correct for languages without inheritance (Go), but incoherent for languages with inheritance that ignore it (TypeScript).

8.8 Mixins with MRO Strictly Dominate Object Composition

The "composition over inheritance" principle from the Gang of Four (1994) has become software engineering dogma. We demonstrate this principle is incorrect for behavior extension in languages with explicit MRO.

8.8.1 Formal Model: Mixin vs Composition **Definition 8.1 (Mixin).** A mixin is a class designed to provide behavior via inheritance, with no standalone instantiation. Mixins are composed via the bases axis, resolved deterministically via MRO.

```
# Mixin: behavior provider via inheritance
class LoggingMixin:
    def process(self):
        print(f"Logging: {self}")
        super().process()

class CachingMixin:
    def process(self):
        if cached := self._check_cache():
            return cached
        result = super().process()
        self._cache(result)
        return result

# Composition via bases (single decision point)
class Handler(LoggingMixin, CachingMixin, BaseHandler):
    pass  # MRO: Handler $\rightarrow$ Logging $\rightarrow$ Caching $\rightarrow$ Base
```

Definition 8.2 (Object Composition). Object composition delegates to contained objects, with manual call-site dispatch for each behavior.

```
# Composition: behavior provider via delegation
class Handler:
    def __init__(self):
        self.logger = Logger()
        self.cache = Cache()

    def process(self):
        self.logger.log(self)  # Manual dispatch point 1
        if cached := self.cache.check():  # Manual dispatch point 2
            return cached
        result = self._do_process()
        self.cache.store(key, result)  # Manual dispatch point 3
        return result
```

8.8.2 Capability Analysis **What composition provides:** 1. [PASS] Behavior extension (via delegation) 2. [PASS] Swapping implementations at runtime 3. [PASS] Multiple behaviors combined

What mixins provide: 1. [PASS] Behavior extension (via super() linearization) 2. [PASS] Swapping implementations at definition time 3. [PASS] Multiple behaviors combined 4. [PASS] **Deterministic conflict resolution** (C3 MRO) — **composition cannot provide** 5. [PASS] **Single decision point** (class definition) — **composition has n call sites** 6. [PASS] **Provenance via MRO** (which mixin provided this behavior?) — **composition cannot provide** 7. [PASS] **Exhaustive enumeration** (list all mixed-in behaviors via `__mro__`) — **composition cannot provide**

Therefore: **Mixin capabilities ⊃ Composition capabilities** (strict superset).

Theorem 8.1 (Mixin Dominance). For behavior extension in languages with deterministic MRO, mixin composition strictly dominates object composition.

Proof. Let $\mathcal{M}$ = capabilities of mixin composition (inheritance + MRO). Let $\mathcal{C}$ = capabilities of object composition (delegation).

Mixins provide: 1. Behavior extension (same as composition) 2. Deterministic conflict resolution via MRO (composition cannot provide) 3. Provenance via MRO position (composition cannot provide) 4. Single decision point for ordering (composition has n decision points) 5. Exhaustive enumeration via `__mro__` (composition cannot provide)

Therefore $\mathcal{C} \subset \mathcal{M}$ (strict subset). By the same argument as Theorem 3.5 (Strict Dominance), choosing composition forecloses capabilities for zero benefit. ■

8.8.3 Connection to Typing Discipline The parallel to Theorem 3.5 is exact:

Typing Disciplines	Architectural Patterns
Structural typing checks only namespace (shape)	Composition checks only namespace (contained objects)
Nominal typing checks namespace + bases (MRO)	Mixins check namespace + bases (MRO)
Structural cannot provide provenance	Composition cannot provide provenance
Nominal strictly dominates	Mixins strictly dominate

Theorem 8.2 (Unified Dominance Principle). In class systems with explicit inheritance (bases axis), mechanisms using bases strictly dominate mechanisms using only namespace.

Proof. Let B = bases axis, S = namespace axis. Let D_S = discipline using only S (structural typing or composition). Let D_B = discipline using $B + S$ (nominal typing or mixins).

D_S can only distinguish types/behaviors by namespace content. D_B can distinguish by namespace content AND position in inheritance hierarchy.

Therefore capabilities(D_S) ⊂ capabilities(D_B) (strict subset). ■

8.9 Validation: Alignment with Python's Design Philosophy

Our formal results align with Python's informal design philosophy, codified in PEP 20 ("The Zen of Python"). This alignment validates that the abstract model captures real constraints.

"Explicit is better than implicit" (Zen line 2). ABCs require explicit inheritance declarations (`class Config(ConfigBase)`), making type relationships visible in code. Duck typing relies on implicit runtime checks (`hasattr(obj, 'validate')`), hiding conformance assumptions. Our Theorem 3.5 formalizes this: explicit nominal typing provides capabilities that implicit shape-based typing cannot.

"In the face of ambiguity, refuse the temptation to guess" (Zen line 12). Duck typing *guesses* interface conformance via runtime attribute probing. Nominal typing refuses to guess,

requiring declared conformance. Our provenance impossibility result (Corollary 6.3) proves that guessing cannot distinguish structurally identical types with different inheritance.

"Errors should never pass silently" (Zen line 10). ABCs fail-loud at instantiation (`TypeError: Can't instantiate abstract class with abstract method validate`). Duck typing fails-late at attribute access, possibly deep in the call stack. Our complexity theorems (Section 4) formalize this: nominal typing has O(1) error localization, while duck typing has $\Omega(n)$ error sites.

"There should be one– and preferably only one –obvious way to do it" (Zen line 13). Our decision procedure (Section 2.5.1) provides exactly one obvious way: in greenfield with inheritance, use nominal typing.

Historical validation: Python's evolution confirms our theorems. Python 1.0 (1991) had only duck typing. Python 2.6 (2007) added ABCs because duck typing was insufficient for large codebases. Python 3.8 (2019) added Protocols for retrofit scenarios. This evolution from duck $\rightarrow$ nominal $\rightarrow$ hybrid exactly matches our formal predictions.

8.10 Connection to Gradual Typing

Our results connect to the gradual typing literature (Siek & Taha 2006, Wadler & Findler 2009). Gradual typing addresses the *retrofit* case: adding types to existing untyped code. Our theorems address the *greenfield* case: choosing types for new code.

The complementary relationship:

Scenario	Gradual Typing	Our Theorems
Retrofit (existing code)	[PASS] Applicable	[WARN] Concession
Greenfield (new code)	[WARN] Overkill	[PASS] Applicable

Gradual typing's insight: When retrofitting types onto untyped code, you cannot mandate inheritance. Structural typing (via the dynamic type `?`) allows gradual migration.

Our insight: When writing new code with inheritance available, structural typing forecloses capabilities. Nominal typing is correct.

The unified view: Gradual typing and nominal typing are not competing paradigms. They address different development contexts: - Use gradual typing to add types to legacy code (retrofit) - Use nominal typing for new code with inheritance (greenfield)

Theorem 8.3 (Gradual-Nominal Complementarity). Gradual typing and nominal typing are complementary, not competing. Gradual typing is correct for retrofit; nominal typing is correct for greenfield.

Proof. Gradual typing's dynamic type `?` allows structural compatibility with untyped code. This is necessary for retrofit (Theorem 3.1: structural typing is valid when bases are unavailable). Nominal typing's isinstance checks require explicit inheritance. This is correct for greenfield (Theorem 3.5: nominal strictly dominates when bases are available). The two disciplines apply to disjoint contexts. ■

---

9. Conclusion

We have presented a methodology for typing discipline selection in object-oriented systems:

1. **The greenfield-retrofit distinction**: Duck typing is retrofit tooling—appropriate when integrating components without shared base classes. Nominal typing is greenfield tooling—mandatory when you control the class hierarchy. This is not a style choice; it is a correctness criterion (Theorem 3.4).
2. **Measurable code quality metrics**: Four metrics derived from the formal model (duck typing density, nominal typing ratio, provenance capability, resolution determinism) enable automated detection of typing discipline violations in codebases.
3. **Formal foundation**: Nominal typing achieves O(1) error localization versus duck typing's $\Omega(n)$ (Theorem 4.3). Duck typing cannot provide provenance because structurally equivalent objects are indistinguishable by definition (Corollary 6.3, machine-checked in Lean 4).
4. **13 case studies demonstrating methodology application**: Each case study identifies the indicators (provenance requirement, MRO-based resolution, type identity as key) that determine which typing discipline is correct. Measured outcomes include elimination of scattered `hasattr()` checks when migrating from duck typing to nominal contracts.
5. **Recurring architectural patterns**: Six patterns require nominal typing: metaclass auto-registration, bidirectional type registries, MRO-based priority resolution, runtime class generation with lineage tracking, descriptor protocol integration, and discriminated unions via `__subclasses__()`.

The methodology in one sentence: If your system requires knowing *which type* provided a value (provenance), or uses inheritance to determine priority (MRO), or needs types as dictionary keys (identity)—use nominal typing. If you're integrating components you don't control—structural typing is an acceptable concession. Shape-based typing is never correct for greenfield code.

The Debate Is Over

For decades, typing discipline has been treated as style. "Pythonic" duck typing versus "Java-style" nominal typing, with structural typing positioned as the modern middle ground. This framing is wrong.

The decision procedure does not output "nominal is preferred." It outputs "nominal is required" or "shape-based typing is a concession." There is no case where duck typing or structural typing is the correct choice for greenfield code with provenance requirements.

Two architects examining identical requirements will derive identical discipline choices. Disagreement indicates incomplete requirements or incorrect procedure application—not legitimate difference of opinion. The question of typing discipline is settled by derivation, not preference.

9.2 Future Work: Cross-Language Validation

Our theorems make falsifiable predictions for other languages. We invite the community to validate or refute these predictions:

Prediction 1 (Java/Spring). Spring Framework's dependency injection should exhibit nominal typing patterns. Specifically: - Bean registration should use `Class<?>` as keys (type identity) -

Autowiring should use `instanceof` checks (nominal subtyping) - Aspect-oriented programming should use MRO-equivalent dispatch

Falsification criterion: If Spring's core DI uses structural matching (interface signature comparison) rather than nominal identity, our Theorem 3.5 is falsified for Java.

Prediction 2 (Ruby/Rails). Rails' ActiveRecord should exhibit nominal typing patterns: - Model inheritance should use `ancestors` for MRO-based dispatch - Polymorphic associations should use `is_a?` checks - Concern composition should use mixin patterns with deterministic ordering

Falsification criterion: If Rails uses duck typing (`respond_to?`) for core model dispatch rather than nominal checks, our theorems are falsified for Ruby.

Prediction 3 (C#/.NET). ASP.NET Core's middleware pipeline should exhibit nominal patterns: - Middleware registration should use `Type` as keys - Dependency injection should use `GetType().IsAssignableFrom()` - Configuration binding should use inheritance hierarchies

Falsification criterion: If ASP.NET Core uses structural matching for middleware dispatch, our theorems are falsified for C#.

Prediction 4 (Go). Go frameworks should exhibit structural patterns (correctly, per Theorem 3.1): - Interface satisfaction should be implicit (no `implements` keyword) - No MRO-based dispatch (Go has no inheritance) - Type identity should be less common than interface satisfaction

Falsification criterion: If Go frameworks extensively use type identity (`reflect.TypeOf`) for dispatch, our Theorem 3.1 (structural is correct for reduced systems) is falsified.

Validation methodology: 1. Clone framework source code 2. Count occurrences of nominal patterns (`isinstance`, `type()`, `__mro__`, `Class<?>`, `is_a?`) 3. Count occurrences of structural patterns (`hasattr`, `respond_to?`, interface matching) 4. Calculate NTR (Nominal Typing Ratio) per Section 8.5 5. Compare to predictions

Expected outcomes: - Java/Spring: NTR > 0.8 (strongly nominal) - Ruby/Rails: NTR > 0.7 (nominal with some duck typing at boundaries) - C#/.NET: NTR > 0.8 (strongly nominal) - Go: NTR < 0.3 (structural, correctly per Theorem 3.1)

These predictions are falsifiable. If the data contradicts our predictions, our theorems are wrong. This is the scientific method applied to programming language theory.

---

10. References

1. Barrett, K., et al. (1996). "A Monotonic Superclass Linearization for Dylan." OOPSLA.
2. Van Rossum, G. (2002). "Unifying types and classes in Python 2.2." PEP 253.
3. The Python Language Reference, §3.3.3: "Customizing class creation."
4. Malayeri, D. & Aldrich, J. (2008). "Integrating Nominal and Structural Subtyping." ECOOP.
5. Malayeri, D. & Aldrich, J. (2009). "Is Structural Subtyping Useful? An Empirical Study." ESOP.
6. Abdelgawad, M. & Cartwright, R. (2014). "NOOP: A Domain-Theoretic Model of Nominally-Typed OOP." ENTCS.
7. Abdelgawad, M. (2016). "Why Nominal-Typing Matters in OOP." arXiv:1606.03809.
8. Gil, J. & Maman, I. (2008). "Whiteoak: Introducing Structural Typing into Java." OOPSLA.

9. Veldhuizen, T. (2006). “Tradeoffs in Metaprogramming.” ACM Computing Surveys.
10. Damaševičius, R. & Štuikys, V. (2010). “Complexity Metrics for Metaprograms.” Information Technology and Control.
11. Liskov, B. & Wing, J. (1994). “A Behavioral Notion of Subtyping.” ACM TOPLAS.
12. Blum, M. (1967). “On the Size of Machines.” Information and Control.
13. Cook, W., Hill, W. & Canning, P. (1990). “Inheritance is not Subtyping.” POPL.
14. de Moura, L. & Ullrich, S. (2021). “The Lean 4 Theorem Prover and Programming Language.” CADE.
15. Leroy, X. (2009). “Formal verification of a realistic compiler.” Communications of the ACM.
16. Klein, G., et al. (2009). “seL4: Formal verification of an OS kernel.” SOSP.
17. Siek, J. & Taha, W. (2006). “Gradual Typing for Functional Languages.” Scheme and Functional Programming Workshop.
18. Wadler, P. & Findler, R. (2009). “Well-Typed Programs Can’t Be Blamed.” ESOP.
19. Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994). “Design Patterns: Elements of Reusable Object-Oriented Software.” Addison-Wesley.
20. Peters, T. (2004). “PEP 20 – The Zen of Python.” Python Enhancement Proposals.
21. TypeScript GitHub Issue #202 (2014). “Nominal types.” https://github.com/microsoft/TypeScript/issues/202
22. TypeScript GitHub Issue #33038 (2019). “Proposal: Nominal Type Tags.” https://github.com/microsoft/Type