Impossibility Theorems for Fixed-Axis Classification Systems: With Application to Type Theory

Tristan Simas
McGill University
tristan.simas@mail.mcgill.ca

February 20, 2026

Abstract

The Problem. Classification systems (type systems, ontologies, taxonomies, schemas) operate over fixed sets of classification axes. We prove this architectural choice has unavoidable consequences: *any fixed-axis system is incomplete for some domain*. This is not a limitation of specific implementations; it is an information-theoretic impossibility.

The Core Theorems (machine-checked, 0 sorries):

- **Fixed Axis Incompleteness:** For any axis set A and any axis $a \notin A$, there exists a domain D that A cannot serve. The information required to answer D's queries does not exist in A.
- **Parameterized Immunity:** For any domain D, there exists an axis set A_D that is complete for D. This set is computable: $A_D = \bigcup_{q \in D} \text{requires}(q)$.
- **The Asymmetry:** Fixed systems guarantee failure for some domain. Parameterized systems guarantee success for all domains. One dominates the other absolutely.
- **Semantic Non-Embeddability:** Disciplines with fewer axes cannot embed disciplines with more axes. The gap is information-theoretic: no encoding, simulation, or workaround can bridge it.
- **Minimality $\Rightarrow$ Orthogonality:** Every minimal complete axis set is orthogonal. Orthogonality is not imposed; it is *derived* from minimality.
- **Matroid Structure:** Orthogonal axis sets satisfy the matroid exchange property. Consequence: all minimal complete sets have equal cardinality (dimension is well-defined).

The Prescriptive Force. These are not design recommendations. They are mathematical necessities:

1. Given any domain D, the required axes are *computable*, not chosen.
2. Missing axes cause *impossibility*, not difficulty. No implementation overcomes a missing axis.
3. The choice of axis-parameterization is *forced* by the requirement of domain-agnosticism.

Application to Type Systems. We instantiate the framework to programming language type systems, proving:

- (B, S) (Bases and Namespace) is the unique minimal complete axis set for *the domain of standard class semantics*. It is still incomplete for domains requiring additional axes.

- (B, S, H) extends this for hierarchical configuration systems. H (hierarchy) was derived when OpenHCS required scope-chain queries that (B, S) could not answer.
- B-inclusive systems strictly dominate B-exclusive systems when $B \neq \emptyset$ (nominal typing dominates structural typing)
- Structural typing cannot embed nominal typing—the four capabilities (provenance, identity, enumeration, conflict resolution) are information-theoretically absent from $\{S\}$-only systems
- Systems using only $\{S\}$ require $\Omega(n)$ error localization; systems using $\{B, S\}$ achieve $O(1)$

The Broader Claim. The impossibility theorems apply to *any* classification system with fixed dimensions, not only type systems. Biological taxonomies, library classification schemes, database schemas, knowledge graphs: all are subject to the same constraints. A fixed set of axes guarantees domains that cannot be served.

Corollary (Incoherence of Preference). Claiming "classification system design is a matter of preference" while accepting the uniqueness theorem instantiates $P \wedge \neg P$. Uniqueness entails $\neg\exists$ alternatives; preference presupposes $\exists$ alternatives. The mathematics admits no choice.

All proofs in Lean 4 (6300+ lines, 190+ theorems, 0 `sorry`).

Keywords: classification theory, impossibility theorems, matroid theory, type systems, formal verification, epistemology

1 Introduction

1.1 The Classification Problem

Every formal system that classifies entities faces a fundamental architectural decision: which dimensions (axes) of classification to employ. Type systems choose between nominal and structural axes. Ontologies choose between taxonomic and mereological axes. Database schemas choose between relational and hierarchical axes.

This paper proves that this choice has unavoidable mathematical consequences. Specifically:

1. **Fixed axis sets guarantee incompleteness.** For any axis set A, there exist domains requiring axes outside A. No implementation within A can answer these domains' queries; the information does not exist.

2. **Parameterized axis sets achieve universality.** A classification system that parameterizes over axes can serve any domain by instantiating the required axes.

3. **The asymmetry is absolute.** Fixed systems fail for some domain (guaranteed). Parameterized systems succeed for all domains (guaranteed). This is not a tradeoff; it is strict dominance.

1.2 Scope: From Type Theory to Classification Theory

This work establishes a general framework for classification systems, proving that axis sets are derivable from domain requirements. The type system application ("nominal vs structural typing") is one instantiation among many.

The framework rests on three core concepts:

- Domains as sets of queries requiring specific information
- Axes as orthogonal dimensions of classification

- Completeness as the ability to answer all queries in a domain

These concepts apply to any classification scheme (type systems, ontologies, taxonomies, schemas). We present results at two levels:

1. **The universal claim** (Section 3): Classification systems with fixed axes are incomplete for some domain. Axis sets are computable from domain requirements.

2. **The type system instantiation** (Sections 4–5): For programming languages with inheritance, (B, S) is the unique minimal complete axis set. Traditional typing discipline names ("nominal," "structural") denote specific axis configurations.

The impossibility theorems are universal; the type system debates are a consequence of wrongly framing axis derivation as preference.

1.3 Metatheoretic Foundations

This work follows the tradition of Liskov & Wing [21], who formalized correctness criteria for subtyping. Where Liskov & Wing asked "what makes subtyping *correct*?", we ask a more fundamental question: "what makes *any classification scheme* complete for its domain?"

Our contribution is not recommending specific classification axes, but proving that the choice is *determined* by domain requirements. We develop a framework where:

- The required axes are *computable* from domain queries

- Missing axes cause *impossibility*, not difficulty

- All minimal complete axis sets have *equal cardinality* (dimension is well-defined)

1.4 Overview

All results are machine-checked in Lean 4 (6300+ lines, 190+ theorems, 0 `sorry` placeholders).

We develop a metatheory applicable to any classification system. The core insight: every such system is characterized by which axes it employs. For type systems, these are (B, S) (Bases and Namespace); for hierarchical configuration, (B, S, H) adds a Scope axis. These axes form a lattice: $\emptyset < S < (B, S) < (B, S, H)$, where each increment strictly dominates the previous.

The pay-as-you-go principle: Each axis increment adds capabilities without cognitive load increase until those capabilities are invoked. Duck typing uses $\emptyset$; structural uses S; nominal uses (B, S) with the same API; scoped resolution uses (B, S, H) with one optional parameter.

The model formalizes what programmers intuitively understand but rarely make explicit:

1. **Axis inclusion requirements** (Theorem 3.4): For languages with explicit inheritance (bases axis $B \neq \emptyset$), systems including $\{B, S\}$ Pareto-dominate systems using only $\{S\}$ (provides strictly more capabilities with zero tradeoffs). Systems excluding B lose nothing only when $B = \emptyset$ by design (e.g., Go); otherwise it is a capability sacrifice, not an alternative. The axis set is **derived** from domain requirements, not preference. In type system terminology: "nominal typing" $= \{B, S\}$; "structural typing" $= \{S\}$.
2. **Complexity separation** (Theorem 4.3): Systems using $\{B, S\}$ achieve O(1) error localization; systems using only $\{S\}$ incoherently (duck typing) require $\Omega(n)$ call-site inspection.
3. **Provenance impossibility** (Corollary 6.3): Systems using only the $\{S\}$ axis cannot answer "which type provided this value?" because structurally equivalent objects are indistinguishable by definition. The B axis is required. Machine-checked in Lean 4.

These theorems yield four measurable code quality metrics:

Metric	What it measures	Indicates
Duck typing density	`hasattr()` per KLOC	Discipline violations (duck typing is incoherent per Theorem 2.10d; other `getattr()` / `AttributeError` patterns may be valid metaprogramming)
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 $\Omega(n)$ error localization

The methodology is validated through case studies from OpenHCS [35], 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 [36]) eliminated 47 scattered `hasattr()` checks and consolidated dispatch logic into explicit ABC contracts.

1.5 Contributions

This paper makes five contributions, ordered from most general to most specific:

1. Impossibility Theorems for Fixed Classification (Section 3):

- **Fixed Axis Incompleteness:** Any classification system with fixed axes is incomplete for some domain. This is information-theoretic: the data required to answer the domain's queries does not exist in the fixed axis set.
- **Uniqueness of Minimal Complete Sets:** All minimal complete axis sets for a domain have equal cardinality. Dimension is well-defined for classification problems.
- **Minimality $\Rightarrow$ Orthogonality:** Every minimal complete axis set is orthogonal. Redundant (non-orthogonal) systems are never minimal. This is a *theorem*, not an assumption.
- **Matroid Structure (Theorem 2.32):** Orthogonal axis sets satisfy the matroid exchange property. This gives basis equicardinality and makes dimension well-defined.
- **Semantic Non-Embeddability (Theorem 3.13a):** Disciplines with fewer axes cannot embed disciplines with more axes. Structural typing cannot simulate nominal typing—the information is simply absent.
- **Axis-Parametric Dominance:** Systems that derive axes from requirements strictly dominate fixed-axis systems. The gap is not a tradeoff; it is absolute.

These theorems apply to *any* classification system: type systems, ontologies, taxonomies, schemas.

2. Type System Instantiation (Sections 4–5):

- **Theorem 3.32 (Model Completeness):** (B, S) captures all runtime-available type information for class-based systems.
- **Theorem 3.13 (Provenance Impossibility):** No system excluding B can compute provenance. This is information-theoretic impossibility.
- **Theorem 3.13a (Non-Embeddability):** Nominal typing cannot be embedded into structural typing. The information gap is absolute, not a matter of encoding.

- **Theorem 3.24 (Complexity Lower Bound):** Systems using only $\{S\}$ incoherently require $\Omega(n)$ error localization. Systems using $\{B, S\}$ achieve $O(1)$. The gap is unbounded.
- **Theorem 3.5:** Systems including $\{B, S\}$ strictly dominate systems using only $\{S\}$ when $B \neq \emptyset$. (In type system terminology: nominal typing dominates structural typing.)

3. Axis-Parametric Extension:

- **Theorem 3.82 (Axis Capability Monotonicity):** Adding an independent axis strictly increases capabilities.
- **Theorem 3.85 (Completeness Uniqueness):** For any domain, the minimal complete axis set is unique.
- **Theorem 3.86 (Axis Derivation Algorithm):** The minimal axis set is computable from domain requirements.
- **OpenHCS instantiation:** For hierarchical configuration (scope chains), a third axis H is derived. The result (B, S, H) enables click-to-provenance across windows.

4. Metatheoretic Foundations (Section 2):

- The axis lattice framework for capability analysis
- Theorem 2.15 (Axis Lattice Dominance): capability monotonicity under axis subset ordering
- The pay-as-you-go principle: axes add capability without cost until invoked

5. Machine-checked Verification (Section 6):

- 6300+ lines of Lean 4 proofs across eleven modules
- 190+ theorems/lemmas with **zero `sorry` placeholders**
- Formalized $O(1)$ vs $\Omega(n)$ complexity separation with adversary-based lower bound
- Universal extension to 8 languages (Java, C#, Rust, TypeScript, Kotlin, Swift, Scala, C++)

6. Empirical Validation (Section 5):

- 13 case studies from OpenHCS (45K LoC production Python codebase)
- Demonstrates theoretical predictions align with real-world architectural decisions
- Four derivable code quality metrics (DTD, NTR, PC, RD)

1.5.1 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):

- `@auto_create_decorator` → `@global_pipeline_config` cascade: 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)

1.5.2 Decision Procedure, Not Preference

The contribution of this paper is not the theorems alone, but their consequence: axis set determination becomes a decision procedure (Theorem 2.26). Given requirements, the axes are derived.

Implications:

1. **Pedagogy.** Architecture courses should not teach "pick the style that feels Pythonic." They should teach how to derive the required axes from domain queries. This is engineering, not taste.
2. **AI code generation.** LLMs can apply the decision procedure. "Given requirements R, apply Algorithm 2.26, emit code with the derived axis set" is an objective correctness criterion. The model either applies the procedure correctly or it does not.
3. **Language design.** Future languages could enforce axis inclusion based on declared requirements. A `@requires_provenance` annotation could mandate B inclusion at compile time.
4. **Formal constraints.** When requirements include provenance, the mathematics constrains the choice: systems excluding B cannot provide this capability (Theorem 3.13, information-theoretic impossibility). The procedure derives the axis set from requirements.

1.5.3 Scope and Limitations

This paper makes absolute claims. We do not argue that including all axes is "preferred" or "more elegant." We prove:

1. **Systems excluding B cannot provide provenance.** Disciplines using only $\{S\}$ check type *shape*: attributes, method signatures. Provenance requires type *identity* from the inheritance graph. Systems excluding B cannot provide what they do not track.
2. **When B $\neq \emptyset$, $\{B, S\}$ systems dominate among fixed-axis systems.** Including both axes provides strictly more capabilities. Adapters eliminate the retrofit exception (Theorem 2.10j). When inheritance exists, $\{B, S\}$ inclusion is the capability-maximizing choice *within the fixed-axis model.*
3. **Excluding required axes is a capability sacrifice.** Systems using only $\{S\}$ (Protocol, structural typing) discard the Bases axis. This eliminates four capabilities (provenance, identity, enumeration, conflict resolution) without providing any compensating capability (a dominated choice when $B \neq \emptyset$).

Fixed-axis vs. axis-parametric correctness. The above results establish dominance *within* fixed-axis type systems. However, any fixed-axis system is fundamentally incomplete (Theorem 3.77, Fixed Axis Incompleteness): there exist domains requiring axes outside the fixed set. The truly *correct* approach is axis-parametric, deriving the minimal axis set from domain requirements (Section 3.6). Fixed-axis maximization (choosing $\{B, S\}$ over $\{S\}$ when $B \neq \emptyset$) is coherent only when the axis-parametric option is unavailable. Put simply: axis-parametric is correct; fixed-axis $\{B, S\}$ is less wrong than fixed-axis $\{S\}$; but both fixed options remain incomplete.

Boundary scope (pulled forward for clarity): when $B = \emptyset$ (no user-declared inheritance), e.g., pure JSON/FFI payloads or languages intentionally designed without inheritance, $\{S\}$-only systems (structural typing) are the coherent choice. Whenever $B \neq \emptyset$ and inheritance metadata is accessible, including B in the axis set strictly dominates excluding it. Systems where $B = \emptyset$ are not exceptions—they simply do not require the B axis, and thus $\{S\}$-only systems lose nothing (but gain nothing either).

The requirements determine the axes; the axes determine the capabilities. Systems requiring provenance cannot exclude B. This is not a design recommendation but a mathematical constraint.

1.6 Roadmap

Section 2: Metatheoretic foundations — The two-axis model (B, S) with names as sugar, abstract class system formalization, and the Axis Lattice Metatheorem (Theorem 2.15)

Section 3: Universal dominance — Strict dominance (Theorem 3.5), information-theoretic completeness (Theorem 3.19), retrofit exception eliminated (Theorem 2.10j)

Section 4: Decision procedure — Deriving typing discipline from system properties
Section 5: Empirical validation — 13 OpenHCS case studies validating theoretical predictions
Section 6: Machine-checked proofs — Lean 4 formalization (6300+ lines)
Section 7: Related work — Positioning within PL theory literature
Section 8: Extensions — Mixins vs composition (Theorem 8.1), TypeScript coherence analysis (Theorem 8.7), gradual typing connection, Zen alignment
Section 9: Conclusion — Implications for PL theory and practice

1.7 Anticipated Objections

Before proceeding, we address objections readers are likely forming. Each is refuted in detail in Appendix A; here we summarize the key points.

"The (B, S) model doesn't capture real type systems." The model is complete by construction. In Python, `type(name, bases, namespace)` is the universal type constructor—a type *is* (B, S), not merely has it. The syntactic name (`__name__`) is metadata stored in the namespace (S axis), not a separate axis. Properties like `__mro__` and `__module__` are *derived from* (B, S), not independent axes. This is definitional closure: no third axis can exist (Theorem 3.5, Appendix A.2).

"Duck typing is more flexible than nominal typing." No. Nominal typing provides a strict superset of duck typing capabilities. Duck typing's "acceptance" of structurally-equivalent types is not a capability—it is the *absence* of the capability to distinguish them. "Flexibility" is the absence of discrimination, not the presence of power (Theorems 3.34–3.36, Appendix A.2).

"The axiom that shape-based typing can't distinguish same-shape types is circular." The axiom is *definitional*, not assumptive. Shape-based typing is *defined* as the discipline that treats types identically when their shapes match. This is what "shape-based" means. The theorems follow from the definition (Lemma 3.5, Appendix A.2).

"A clever extension could recover provenance for duck typing." No. Theorem 3.5 proves that any computable extension over $\{S\}$ (even with type names) cannot recover provenance. The common response "just check `type(x)`" proves the point: inspecting `type(x)` consults the B axis. Once you use B, you have adopted nominal typing (Appendix A.2).

"You only tested on one codebase (OpenHCS)." This conflates existential witnesses with premises. The dominance theorems are proven from the *definition* of shape-based typing—no codebase is referenced in the proofs. OpenHCS demonstrates that the four capabilities (provenance, identity, enumeration, conflict resolution) are *achievable*, not that the theorems are true. Analogy: proving $\Omega(n \log n)$ for comparison sorts doesn't require testing multiple arrays (Appendix A.2).

"Dominance implies you should migrate all legacy codebases." No. Theorem 3.5 formally proves that Pareto dominance does NOT imply beneficial migration. Dominance is codebase-independent; migration cost is codebase-dependent. The theorem characterizes *capabilities*, not *recommendations* (Appendix A.2).

"You just need code review and discipline, not language features." Discipline *is* the external oracle. When DOF > 1, consistency requires an oracle to resolve disagreements. "Code review" is exactly that oracle—human-maintained, fallible, bypassable. Language enforcement cannot be forgotten; human discipline can (Appendix A.2).

"The proofs are trivial—just `rfl`." When modeling is correct, theorems become definitional. This is a feature. Not all proofs are `rfl`: `provenance_impossibility` requires 40+ lines of actual reasoning (assumption → lemma application → contradiction). The contribution is making the right definitions so that consequences follow structurally (Appendix A.2).

"This is just subtyping." No. Subtyping asks: "Can I use type A where type B is expected?" This is a *relationship* between types. Our claim is about type *construction*: extending `type(name, bases, namespace)` to `type(name, bases, namespace, **axes)`. These are orthogonal concerns. Subtyping operates on types that already exist; axis-parametric construction determines type *identity* at creation time. Whether $T(\text{scope} = X) <: T(\text{scope} = Y)$ is a question that comes *after* our proposal—and we make no claims about it (Appendix A.2).

If you have an objection not listed above, check Appendix A (17 concerns addressed) before concluding it has not been considered.

---

2 Preliminaries

2.1 Definitions

Definition 2.1 (Class). A class C is a triple (name, bases, namespace) where: - name ∈ String — the identity of the class - bases ∈ List[Class] — explicit inheritance declarations - namespace ∈ 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 ($\{B, S\}$ Typing). x satisfies A iff A ∈ MRO(type(x)). The constraint is checked via explicit inheritance.

Remark: In type system terminology, this is called "nominal typing."

Definition 2.4 ($\{S\}$-Only Typing with Declared Interfaces). x satisfies A iff namespace(x) ⊇ signature(A). The constraint is checked via method/attribute matching. In Python, `typing.Protocol` implements this: a class satisfies a Protocol if it has matching method signatures, regardless of inheritance.

Remark: In type system terminology, this is called "structural typing."

Definition 2.5 ($\{S\}$-Only Incoherent 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.

Remark: In type system terminology, this incoherent pattern is called "duck typing."

Observation 2.1 ($\{S\}$-Only Systems Are Shape-Based). $\{S\}$-only systems (with or without declared interfaces) are both *shape-based*: they check what methods or attributes an object has, not what type it is. $\{B, S\}$ 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 $\{S\}$-only typing cannot provide provenance—regardless of whether the shape-checking happens at compile time (with declared interfaces) or runtime (incoherent).

Remark: In type system terminology, $\{S\}$-only with declared interfaces is "structural typing"; $\{S\}$-only incoherent is "duck typing"; $\{B, S\}$ is "nominal typing."

Complexity distinction: While $\{S\}$-only systems with and without declared interfaces are both shape-based, they differ critically in *when* the shape-checking occurs:

- **$\{S\}$-only with declared interfaces** (Protocol, structural typing): Shape-checking at *static analysis time* or *type definition time*. Complexity: O(k) where k = number of classes implementing the protocol.
- **$\{S\}$-only incoherent** (hasattr/getattr, duck typing): Shape-checking at *runtime, per call site*. Complexity: Ω(n) where n = number of call sites.

This explains why $\{S\}$-only with declared interfaces (TypeScript interfaces, Go interfaces, Python Protocols) is considered superior to incoherent use in practice: both are shape-based, but declared interfaces perform the checking once at compile/definition time, while incoherent use repeats the checking at every usage site.

Critical insight: Even though $\{S\}$-only with declared interfaces has better complexity than incoherent use (O(k) vs Ω(n)), *both* are strictly dominated by $\{B, S\}$ typing's O(1) error localization (Theorem 4.1). $\{B, S\}$ typing checks inheritance at the single class definition point—not once per implementing class ($\{S\}$-only declared) or once per call site ($\{S\}$-only incoherent).

Remark: In type system terminology, $\{B, S\}$ is nominal typing; $\{S\}$-only with declared interfaces is structural typing; incoherent is duck typing.

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` is metadata stored in the namespace (as `__name__`), not an independent axis. It carries no semantic weight for typing purposes.

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 Axes **Definition 2.7 (Abstract Class System).** A class system is a tuple (B, S) where: - B: Bases — the set of explicitly declared parent types (inheritance). The B axis provides access to the full inheritance chain, including the type itself and all ancestors via transitive closure. - S: Namespace — the set of (attribute, value) pairs defining the type's interface.

Remark (Syntactic Names Are Not an Axis). Language-level type names (`__name__` in Python, `getName()` in Java) are *not* part of the semantic model. They are metadata assigned at definition time for debugging, serialization, and error messages. The semantic identity of a type—what determines its capabilities and compatibility—is fully captured by (B, S). Two types with identical (B, S) but different syntactic names are semantically equivalent: they support the same operations, have the same inheritance relationships, and are indistinguishable to any typing discipline. We do not include N as an axis because it adds no typing capability.

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).

Remark (Implicit Root Classes). In Python, every class implicitly inherits from `object`: `class X: pass` has `X.__bases__ == (object,)`. Definition 2.9's "$B = \emptyset$" refers to the abstract model where inheritance from a universal root (Python's `object`, Java's `Object`) is elided. Equivalently, $B = \emptyset$ means "no user-declared inheritance beyond the implicit root." The theorems apply when $B \neq \emptyset$ in this sense—i.e., when the programmer explicitly declares inheritance relationships.

Remark (Go Embedding $\neq$ Inheritance). Go's struct embedding provides method forwarding but is not inheritance: (1) embedded methods cannot be overridden—calling `outer.Method()` always invokes the embedded type's implementation, (2) there is no MRO—Go has no linearization algorithm, (3) there is no `super()` equivalent. Embedding is composition with syntactic sugar, not polymorphic inheritance. Therefore Go has $B = \emptyset$.

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.

Remark (Operational Characterization). In Python, shape-based compatibility reduces to capability probing via `hasattr`: `all(hasattr(x, a) for a in S(T))`. We use `hasattr` (not `getattr`) because shape-based typing is about *capability detection*, not attribute retrieval. `getattr` involves metaprogramming machinery (`__getattr__`, `__getattribute__`, descriptors) orthogonal to type discipline.

Remark (Partial vs Full Structural Compatibility). Definition 2.10 uses partial compatibility ($\supseteq$): x has *at least* T's interface. Full compatibility ($=$) requires exact match. Both are $\{S\}$-only disciplines; the capability gap (Theorem 2.17) applies to both. The distinction is a refinement *within* the S axis, not a third axis.

Definition 2.10a (Typing Discipline Completeness). A typing discipline is *complete* if it provides a well-defined, deterministic answer to "when is x compatible with T?" for all x and declared T. Formally: there exists a predicate $\text{compatible}(x, T)$ that is well-defined for all (x, T) pairs where T is a declared type constraint.

Remark (Completeness vs Coherence). Definition 2.10a defines *completeness*: whether the discipline answers the compatibility question. Definition 8.3 later defines *coherence*: whether the discipline's answers align with runtime semantics. These are distinct properties. A discipline can be complete but incoherent (TypeScript's structural typing with `class`), or incomplete and thus trivially incoherent (duck typing).

Definition 2.10b (Structural Typing). Structural typing with declared interfaces (e.g., `typing.Protocol` [19, 39]) is coherent: T is declared as a Protocol with interface $S(T)$, and compatibility is $S(\text{type}(x)) \supseteq S(T)$. The discipline commits to a position: "structure determines compatibility."

Definition 2.10c (Duck Typing). Duck typing is ad-hoc capability probing: `hasattr(x, attr)` [31] for individual attributes without declaring T. No interface is specified; the "required interface" is implicit in whichever attributes the code path happens to access.

Theorem 2.10d (Duck Typing Incoherence). Duck typing is not a coherent typing discipline.

Proof. A coherent discipline requires a well-defined $\text{compatible}(x, T)$ for declared T. Duck typing:

1. **Does not declare T.** There is no Protocol, no interface, no specification of required capabilities. The "interface" is implicit in the code.

2. **Provides different answers based on code path.** If module A probes `hasattr(x, 'foo')` and module B probes `hasattr(x, 'bar')`, the same object x is "compatible" with A's requirements iff it has `foo`, and "compatible" with B's requirements iff it has `bar`. There is no unified T to check against.

3. **Commits to neither position on structure-semantics relationship:**

   - "Structure $=$ semantics" would require checking *full* structural compatibility against a declared interface
   - "Structure $\neq$ semantics" would require nominal identity via inheritance
   - Duck typing checks *partial* structure *ad-hoc* without declaration—neither position

A discipline that gives different compatibility answers depending on which code path executes, with no declared T to verify against, is not a discipline. It is the absence of one. ∎

Related work (duck typing formalization). Refinement-based analyses and logics for dynamic languages approximate duck-typed behaviour statically (e.g., [12, 11]) and empirical interface extraction for dynamic checks has been explored [20]. These systems aim to prove safety for specific programs, not to define a globally coherent predicate $\text{compatible}(x, T)$ for undeclared T that is stable across code paths. Our incoherence result concerns that global typing-discipline property (Definition 8.3); it does not deny the usefulness of such analyses for individual programs.

Definition 2.10b (Coherence). A typing discipline is *coherent* iff type compatibility is determined by a declared interface or type, not by ad-hoc call-site probing. Formally: coherence requires a stable compatibility predicate that exists independently of any particular call site.

Corollary 2.10e (Duck Typing vs Structural Typing). Duck typing ($\{S\}$, incoherent) is strictly weaker than structural typing with Protocols ($\{S\}$, coherent). Both use the same axis; the distinction is coherence.

Proof. Protocols declare T, enabling static verification, documentation, and composition guarantees. Duck typing declares nothing. A Protocol-based discipline is coherent (Definition 2.10b); duck typing is not (Theorem 2.10d). ■

Corollary 2.10f (No Valid Context for Duck Typing). There exists no production context where duck typing is the correct choice.

Proof. In systems with inheritance ($B \neq \emptyset$): nominal typing ($\{B, S\}$, coherent) strictly dominates any system with fewer axes (Remark 3.82a). In systems without inheritance ($B = \emptyset$): structural typing with Protocols ($\{S\}$, coherent) strictly dominates incoherent duck typing ($\{S\}$, incoherent). The only "advantage" of duck typing—avoiding interface declaration—is not a capability but deferred work with negative value (lost verification, documentation, composition guarantees).

Note: This establishes dominance *within fixed-axis systems*. The ultimately correct choice is axis-parametric (Remark 3.82b): supporting N axes costs nothing when only $k < N$ are used, but enables extension when requirements grow. ■

Theorem 2.10g (Structural Typing Eliminability). In systems with inheritance ($B \neq \emptyset$), structural typing is eliminable via boundary adaptation.

Proof. Let S be a system using Protocol P to accept third-party type T that cannot be modified.

1. **Adapter construction.** Define adapter class: `class TAdapter(T, P_as_ABC): pass`
2. **Boundary wrapping.** At ingestion, wrap: `adapted = TAdapter(instance)` (for instances) or simply use `TAdapter` as the internal type (for classes)
3. **Internal nominal typing.** All internal code uses `isinstance(x, P_as_ABC)` with nominal semantics
4. **Equivalence.** The adapted system S' accepts exactly the same inputs as S but uses nominal typing internally

The systems are equivalent in capability. Structural typing provides no capability that nominal typing with adapters lacks. ■

Corollary 2.10h (Structural Typing as Convenience). When $B \neq \emptyset$, structural typing (Protocol) is not a typing necessity but a convenience—it avoids writing the 2-line adapter class. Convenience is not a typing capability.

Corollary 2.10i (Typing Discipline Hierarchy). The typing disciplines form a strict hierarchy. Note: N (names) is not an axis—it is syntactic sugar (Theorem 2.2). The axes are B (bases) and S (namespace).

1. **Duck typing** ($\{S\}$, incoherent): No declared interface (Theorem 2.10d). Never valid.
2. **Structural typing** ($\{S\}$, coherent): Declared interface (Protocol). Eliminable when $B \neq \emptyset$ (Theorem 2.10g). Valid only when $B = \emptyset$.
3. **Nominal typing** ($\{B, S\}$, coherent): Declared hierarchy (ABC). The only non-eliminable discipline for systems with inheritance.

Theorem 2.10j (Protocol Is Strictly Dominated When B $\neq \emptyset$). In systems with inheritance, Protocol is strictly dominated by explicit adapters.

Proof. Compare the two approaches for accepting third-party type T:

Property	Protocol	Explicit Adapter
Accepts same inputs	Yes	Yes
Documents adaptation boundary	No (implicit)	Yes (class definition)
Failure mode	Runtime (`isinstance` returns False, or missing method during execution)	Class definition time (if T lacks required methods)
Provenance	No (T not in your hierarchy)	Yes (adapter is in your hierarchy)
Explicit	No	Yes

The adapter provides strictly more: same inputs, plus explicit documentation, plus fail-loud at definition time, plus provenance. Protocol provides strictly less.

Protocol's only "advantage" is avoiding the 2-line adapter class. But avoiding explicitness is not an advantage—it is negative value. "Explicit is better than implicit" (Zen of Python, line 2). ■

Corollary 2.10k (Protocol's Value Proposition Is Negative). When $B \neq \emptyset$, Protocol trades explicitness, fail-loud behavior, and provenance for 2 fewer lines of code. Protocol's value proposition is negative.

Corollary 2.10k' (Protocol Is a Concession, Not an Alternative). When $B \neq \emptyset$, choosing Protocol is a *concession*—accepting reduced capabilities to defer adapter work. It is not an *alternative* because:

1. Protocol provides no capability that ABCs with adapters lack (Theorem 2.10j)
2. ABCs with adapters provide four capabilities Protocol lacks (provenance, identity, enumeration, conflict resolution)
3. The only "benefit" of Protocol is avoiding 2 lines of adapter code
4. Avoiding work is not a capability

An *alternative* implies comparable standing; a *concession* implies acknowledged inferiority for pragmatic expedience. Protocol is the latter. For Python systems where $B \neq \emptyset$, ABCs with adapters is the single non-concession choice.

Corollary 2.10l (Complete Typing Discipline Validity). The complete validity table:

Discipline	When $B \neq \emptyset$	When $B = \emptyset$
Duck typing	Never (incoherent)	Never (incoherent)
Protocol	Never (dominated by adapters)	Valid (only coherent option)
Nominal/Adapters	Always	N/A (requires B)

2.4.2a The Metaprogramming Capability Gap Beyond typing discipline, nominal and structural typing differ in a second, independent dimension: **metaprogramming capability**. This gap is not an implementation accident—it is mathematically necessary.

Definition 2.10m (Declaration-Time Event). A *declaration-time event* occurs when a type is defined, before any instance exists. Examples: class definition, inheritance declaration, trait implementation.

Definition 2.10n (Query-Time Check). A *query-time check* occurs when type compatibility is evaluated during program execution. Examples: `isinstance()`, Protocol conformance check, structural matching.

Definition 2.10o (Metaprogramming Hook). A *metaprogramming hook* is a user-defined function that executes in response to a declaration-time event. Examples: `__init_subclass__()`, metaclass `__new__()`, Rust's `#[derive]`.

Theorem 2.10p (Hooks Require Declarations). Metaprogramming hooks require declaration-time events. Structural typing provides no declaration-time events for conformance. Therefore, structural typing cannot provide conformance-based metaprogramming hooks.

Proof. 1. A hook is a function that fires when an event occurs. 2. In nominal typing, `class C(Base)` is a declaration-time event. The act of writing the inheritance declaration fires hooks: Python's `__init_subclass__()`, metaclass `__new__()`, Java's annotation processors, Rust's derive macros. 3. In structural typing, "Does X conform to interface I?" is evaluated at query time. There is no syntax declaring "X implements I"—conformance is inferred from structure. 4. No declaration $\rightarrow$ no event. No event $\rightarrow$ no hook point. 5. Therefore, structural typing cannot provide hooks that fire when a type "becomes" conformant to an interface. ■

Theorem 2.10q (Enumeration Requires Registration). To enumerate all types conforming to interface I, a registry mapping types to interfaces is required. Nominal typing provides this registry implicitly via inheritance declarations. Structural typing does not.

Proof. 1. Enumeration requires a finite data structure containing conforming types. 2. In nominal typing, each declaration `class C(Base)` registers C as a subtype of Base. The transitive closure of declarations forms the registry. `__subclasses__()` queries this registry in $O(k)$ where $k = |\text{subtypes}(T)|$. 3. In structural typing, no registration occurs. Conformance is computed at query time by checking structural compatibility. 4. To enumerate conforming types under structural typing, one must iterate over all types in the universe and check conformance for each. In an open system (where new types can be added at any time), $|\text{universe}|$ is unbounded. 5. Therefore, enumeration under structural typing is $O(|\text{universe}|)$, which is infeasible for open systems. ∎

Corollary 2.10r (Metaprogramming Capability Gap Is Necessary). The gap between nominal and structural typing in metaprogramming capability is not an implementation choice—it is a logical consequence of declaration vs. query.

Capability	Nominal Typing	Structural Typing	Why
Definition-time hooks	Yes (`__init_subclass__`, metaclass)	No	Requires declaration event
Enumerate implementers	Yes (`__subclasses__()`, O(k))	No ($O(\infty)$ in open systems)	Requires registration
Auto-registration	Yes (metaclass `__new__`)	No	Requires hook
Derive/generate code	Yes (Rust `#[derive]`, Python descriptors)	No	Requires declaration context

Corollary 2.10s (Universal Applicability). This gap applies to all languages:

Language	Typing	Enumerate implementers?	Definition-time hooks?
Go	Structural	No	No
TypeScript	Structural	No	No (decorators are nominal—require `class`)
Python Protocol	Structural	No	No
Python ABC	Nominal	Yes (`__subclasses__()`)	Yes (`__init_subclass__`, metaclass)
Java	Nominal	Yes (reflection)	Yes (annotation processors)
C#	Nominal	Yes (reflection)	Yes (attributes, source generators)
Rust traits	Nominal (`impl`)	Yes	Yes (`#[derive]`, proc macros)
Haskell typeclasses	Nominal (`instance`)	Yes	Yes (deriving, TH)

Remark (TypeScript Decorators). TypeScript decorators appear to be metaprogramming hooks, but they attach to *class declarations*, not structural conformance. A decorator fires when `class C` is defined—this is a nominal event (the class is named and declared). Decorators cannot fire when "some object happens to match interface I"—that is a query, not a declaration.

Remark (The Two Axes of Dominance). Nominal typing strictly dominates structural typing on two independent axes: 1. **Typing capability** (Theorems 2.10j, 2.18): Provenance, identity, enumeration, conflict resolution 2. **Metaprogramming capability** (Theorems 2.10p, 2.10q): Hooks, registration, code generation

Neither axis is an implementation accident. Both follow from the structure of declaration vs. query. Protocol is dominated on both axes.

Remark. Languages without inheritance (Go) have $B = \emptyset$ by design. For these languages, structural typing with declared interfaces is the correct choice—not because structural typing is superior, but because nominal typing requires B and Go provides none. Go's interfaces are coherent ($\{S\}$, declared). Go does not use duck typing.

Remark (Historical Context). Duck typing became established in Python practice without formal capability analysis. This paper provides the first machine-verified comparison of typing discipline capabilities. See Appendix B for additional historical context.

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 2.13 (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 2.14 (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 2.13, provenance requires B. Shape-based typing has no access to B. Therefore shape-based typing cannot provide provenance. ∎

2.4.3a Representational Cost

Definition 2.14a (Query Interface). For a domain D, let Q_D be the set of queries the domain will ever ask of the type system. A query $q \in Q_D$ is a function from types to answers: $q : \text{Type} \to \text{Answer}$.

Definition 2.14b (Adequacy). A representation R is *adequate* for Q_D if every query $q \in Q_D$ factors through R. That is, for each q, there exists f such that $q = f \circ R$.

Definition 2.14c (Representational Cost). The *representational cost* of representation R for query set Q_D is:

$$\text{cost}(R, Q_D) := \min\{\text{bits required by any adequate representation}\}$$

This is an information-theoretic measure: the minimal state/bits needed to answer the declared queries. Human factors (cognitive load, learning curves) are explicitly out of scope.

Theorem 2.14d (No Unnecessary Cost). If an axis X is never used by any query in Q_D, then adding X as a constant field adds zero semantic cost.

Proof. A constant carries zero bits of information. If no query in Q_D inspects X, then the representation with $X := \star$ (constant) answers exactly the same queries as the representation without X. The minimal adequate information is unchanged. ∎

Theorem 2.14e (Necessity of Axes). If there exist types t_1, t_2 such that:

1. t_1 and t_2 are equal under any projection that drops axis X, but
2. some query $q \in Q_D$ returns different answers: $q(t_1) \neq q(t_2)$,

then any adequate representation for Q_D must carry at least one bit distinguishing t_1 from t_2—i.e., the cost is forced by the query set.

Proof. Suppose R is adequate for Q_D and drops X. Then $R(t_1) = R(t_2)$ by hypothesis (1). But adequacy requires $q = f \circ R$, so $q(t_1) = f(R(t_1)) = f(R(t_2)) = q(t_2)$. This contradicts hypothesis (2). Therefore no representation dropping X is adequate. ∎

Corollary 2.14f (Necessity of B). For any domain D that asks provenance queries, the Bases axis B is necessary. The cost of including B is forced by the query set, not a design choice.

Proof. Immediate from Theorem 2.14e with $X = B$: Theorem 2.13 shows provenance requires MRO, which requires B. Types with identical S but different B can have different provenance. ∎

Table 7: Cross-language instantiation of the (B, S) model

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

2.4.4 Cross-Language Instantiation All four languages provide **runtime access to both axes**.

Clarification (Syntactic Names vs. Semantic Identity). In our model, the semantic identity of a type is determined by (B, S). The syntactic name (`__name__` in Python, `getName()` in Java) is *not* computable from (B, S)—it is metadata assigned at definition time. When we say "N is not an axis," we mean: the semantic content that determines type compatibility and capabilities is fully captured by (B, S). Two types with identical (B, S) are semantically equivalent for all typing purposes, even if they have different `__name__` strings. The syntactic name is useful for debugging and serialization but contributes no typing capability beyond what (B, S) provides.

The critical difference lies in which axes the **type system** inspects.

Table 2.2: Generic Types Across Languages — Extended (B, S), Not a Third Axis

Language	Generics	Encoding	Runtime Behavior
Java	`List<T>`	Parameterized N: `(List, [T])`	Erased to `List`
C#	`List<T>`	Parameterized N: `(List, [T])`	Fully reified
TypeScript	`Array<T>`	Parameterized N: `(Array, [T])`	Compile-time only
Rust	`Vec<T>`	Parameterized N: `(Vec, [T])`	Monomorphized
Kotlin	`List<T>`	Parameterized N: `(List, [T])`	Erased (reified via `inline`)
Swift	`Array<T>`	Parameterized N: `(Array, [T])`	Specialized at compile-time
Scala	`List[T]`	Parameterized N: `(List, [T])`	Erased
C++	`vector<T>`	Parameterized N: `(vector, [T])`	Template instantiation

Key observation: No major language invented a third axis for generics. Generic type parameters extend the (B, S) axes: type arguments become part of the namespace S (determining available operations) and may influence B (inheritance from parameterized parents). The (B, S) model is **universal** across generic type systems.

2.5 The Axis Lattice Metatheorem

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

Clarification: N is not an axis. Syntactic names are metadata, not semantic content (see Definition 2.7 remark). The semantic axes are B (bases) and S (namespace). What distinguishes disciplines using the same axes is *coherence*: whether compatibility is determined by a declared contract.

Discipline	Axes	Coherent?	Example
Untyped	$\emptyset$	N/A	Accept all
Duck typing	$\{S\}$	No	`hasattr` probing
Structural (declared)	$\{S\}$	Yes	OCaml row types, `Protocol`
Nominal	$\{B, S\}$	Yes	ABCs, `isinstance`

Critical distinction within $\{S\}$: Duck typing and structural typing (Protocol, OCaml) use the same axis set. The difference is coherence (Definition 2.10b):

Discipline	Axes	Interface Declared?	Coherent?
Duck typing	$\{S\}$	No (ad-hoc `hasattr`)	No (Thm 2.10d)
OCaml structural	$\{S\}$	Yes (inline type)	Yes
Protocol	$\{S\}$	Yes (named interface)	Yes
Nominal	$\{B, S\}$	Yes (class hierarchy)	Yes

Duck typing and OCaml structural typing both use $\{S\}$, but duck typing has **no declared interface**—conformance is checked ad-hoc at runtime via `hasattr`. OCaml declares the interface inline: `< get : int; set : int -> unit >` is a complete type specification, statically verified.

Protocol also uses $\{S\}$. The Protocol *name* is not an axis—it is a label for a declared shape. The difference from duck typing is that Protocol declares the shape upfront (coherent), while duck typing probes ad-hoc (incoherent).

Theorem 2.10d (Incoherence) applies to duck typing, not to Protocol or OCaml. The incoherence arises from the lack of a declared interface, not from using axis subset $\{S\}$.

Theorems 2.10p-q (Metaprogramming Gap) apply to structural typing. Structural disciplines (Protocol, OCaml) cannot enumerate conforming types or provide definition-time hooks, because conformance is checked at use sites, not at a declaration event. This is independent of coherence.

Note: `hasattr(obj, 'foo')` checks namespace membership via the S axis. Our provenance impossibility theorems apply to all $\{S\}$-only disciplines.

Theorem 2.15 (Axis Lattice Dominance). For any axis subsets $A \subseteq A' \subseteq \{B, S\}$, the capabilities of discipline using A are a subset of capabilities of discipline using A':

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

Proof. Each axis enables specific capabilities:

- B: Provenance, identity, enumeration, conflict resolution
- S: Interface checking

Names are labels, not axes (Theorem 2.2). 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 2.16 (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 2.17 (Capability Completeness). The capability set $\mathcal{C}_B = \{\text{provenance, identity, enumeration, conflict resolution}\}$ is **exactly** the set of capabilities enabled by the Bases axis. Formally:

$$c \in \mathcal{C}_B \iff c \text{ requires } B$$

Proof. We prove both directions:

($\Rightarrow$) Each capability in $\mathcal{C}_B$ requires B:

1. **Provenance** ("which type provided value v?"): By Definition 2.12, provenance queries require MRO traversal. MRO is the C3 linearization of ancestors, which is the transitive closure over B. Without B, MRO is undefined. ✓
2. **Identity** ("is x an instance of T?"): By Definition 2.11, nominal compatibility requires $T \in \text{ancestors}(\text{type}(x))$. Ancestors is defined as transitive closure over B. Without B, ancestors is undefined. ✓
3. **Enumeration** ("what are all subtypes of T?"): A subtype S of T satisfies $T \in \text{ancestors}(S)$. Enumerating subtypes requires inverting the ancestor relation, which requires B. ✓
4. **Conflict resolution** ("which definition wins in diamond inheritance?"): Diamond inheritance produces multiple paths to a common ancestor. Resolution uses MRO ordering, which requires B. ✓

(⇐) No other capability requires B:
We exhaustively enumerate capabilities NOT in $\mathcal{C}_B$ and show none require B:

5. **Interface checking** ("does x have method m?"): Answered by inspecting $S(\text{type}(x))$. Requires only S. Does not require B. ✓
6. **Type naming** ("what is the name of type T?"): Answered by inspecting $N(T)$. Requires only N. Does not require B. ✓
7. **Value access** ("what is $x.a$?"): Answered by attribute lookup in $S(\text{type}(x))$. Requires only S. Does not require B. ✓

   Remark (Inherited Attributes). For inherited attributes, $S(\text{type}(x))$ means the *effective* namespace including inherited members. Computing this effective namespace initially requires B (to walk the MRO), but once computed, accessing a value from the flattened namespace requires only S. The distinction is between *computing* the namespace (requires B) and *querying* a computed namespace (requires only S). Value access is the latter.

8. **Method invocation** ("call $x.m()$"): Answered by retrieving m from S and invoking. Requires only S. Does not require B. ✓

No capability outside $\mathcal{C}_B$ requires B. Therefore $\mathcal{C}_B$ is exactly the B-dependent capabilities. ■

Significance: This is a **tight characterization**, not an observation. The capability gap is not "here are some things you lose"—it is "here is **exactly** what you lose, nothing more, nothing less." This completeness result is what distinguishes a formal theory from an enumerated list.

Theorem 2.18 (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 2.14) 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 = {}):
        RETURN structural  # Theorem 3.1: correct when B absent

    # For all cases where B != {}:
    RETURN nominal  # Theorem 2.18: strict dominance
```

```
# Note: "retrofit" is not a separate case. When integrating
# external types, use explicit adapters (Theorem 2.10j).
# Protocol is a convenience, not a correct discipline.
```

This is a **decision procedure**, not a preference. The output is determined by whether $B = \emptyset$.

2.6 Axis-Parametric Classification Theory

The preceding sections establish typing disciplines as axis projections: nominal typing uses $\{B, S\}$, structural typing uses $\{S\}$. This raises a more fundamental question: **how do we determine which axes a domain requires?**

We now develop a general framework where axis sets are *derived* from domain requirements, not chosen from a menu. This transforms typing discipline selection from preference to computation.

2.6.1 Core Definitions

Definition 2.19 (Domain). A *domain* D is a set of queries that a classification system must answer. Each query is a computable function from types to answers.

Definition 2.20 (Axis). An *axis* A is a recursive lattice structure that:

- Has a carrier type with partial order
- Supports recursive self-reference: $A \to \text{List } A$ (e.g., bases have bases)
- Provides capabilities not derivable from other axes

Definition 2.21 (Axis Independence). Axis A is *independent* of axis set $\mathcal{A}$ iff $\exists$ query q such that A can answer q but no projection from $\mathcal{A}$ can answer q.

Definition 2.22 (Completeness). An axis set $\mathcal{A}$ is *complete* for domain D iff every query $q \in D$ is answerable using only axes in $\mathcal{A}$.

Remark 2.22a (Completeness is Domain-Relative). Completeness is always relative to a domain. An axis set can be complete for domain D_1 and incomplete for domain D_2. For example, (B, S) is complete for standard class semantics but incomplete for hierarchical configuration systems requiring scope chains. There is no "absolutely complete" axis set: for any fixed set $\mathcal{A}$, there exists a domain requiring an axis not in $\mathcal{A}$ (Theorem 2.28).

Definition 2.23 (Minimality). An axis set $\mathcal{A}$ is *minimal* for domain D iff:

1. $\mathcal{A}$ is complete for D, and
2. No proper subset of $\mathcal{A}$ is complete for D

2.6.2 Fundamental Theorems

Theorem 2.24 (Axis Capability Monotonicity). For any axis set $\mathcal{A}$ and independent axis X:

$$\text{capabilities}(\mathcal{A} \cup \{X\}) \supsetneq \text{capabilities}(\mathcal{A})$$

Proof. By independence, $\exists q$ that $\mathcal{A}$ cannot answer but X can. Adding X enables q while preserving all existing capabilities. ∎

Theorem 2.25 (Completeness Uniqueness). For any domain D with requirements Q, if $\mathcal{A}_1$ and $\mathcal{A}_2$ are both minimal complete for D, then $\mathcal{A}_1 \cong \mathcal{A}_2$ (isomorphic as axis sets).

Proof. Suppose $\mathcal{A}_1 \neq \mathcal{A}_2$. WLOG $\exists A \in \mathcal{A}_1, A \notin \mathcal{A}_2$. By minimality of $\mathcal{A}_1$, $\exists q$ requiring A. By completeness of $\mathcal{A}_2$, some axis in $\mathcal{A}_2$ answers q. That axis must be isomorphic to A (answers same queries). Contradiction. ∎

Theorem 2.26 (Axis Derivation Algorithm). The minimal complete axis set for domain D is computable:

```
derive_axes(D):
  A := {}
  for q in queries(D):
    if not answerable(A, q):
      A := A union {minimal_axis_for(q)}
  return collapse(A)  -- remove derivable axes
```

The result is unique (by Theorem 2.25), minimal, and complete.

Proof. The algorithm constructs an axis set that is complete (every query is answerable) and minimal (each axis is required by at least one query). By Theorem 2.25, this is the unique minimal complete set. ■

2.6.3 Implications for Typing Disciplines

Corollary 2.27 (Typing Disciplines as Axis Configurations). Traditional typing discipline names are shorthand for axis configurations:

- **Structural typing** = $\{S\}$-only systems (namespace axis only)
- **Nominal typing** = $\{B, S\}$ systems (bases and namespace)
- **Duck typing** = incoherent use of $\{S\}$ (Theorem 2.10d)

The question "nominal or structural?" presupposes a choice exists. Theorem 2.26 shows the axis set is *computable* from domain requirements.

Theorem 2.28 (Fixed-Axis Incompleteness — Preview). For any fixed axis set A and any axis $a \notin A$, there exists a domain D requiring a. Therefore A is incomplete for D.

Proof sketch. Construct domain D with a query q that requires axis a. Since $a \notin A$, the query is unanswerable by A. Full proof in Section 3.11. ■

Remark 2.29 (Reframing the Contribution). This paper's contribution is not "nominal typing is better than structural typing." It is:

1. Proving axes are *derived* from domain requirements (Theorem 2.26)
2. Showing fixed-axis systems are fundamentally incomplete (Theorem 2.28)
3. Demonstrating that type system debates arise from wrongly framing axis derivation as preference

Section 3 establishes these results formally. The type system instantiation ("nominal vs structural") is one application among many.

2.7 Matroid Structure and Axis Independence

The axis framework possesses deep combinatorial structure: **minimal complete axis sets form matroids**. This is not an assumption—it is a theorem, proven mechanically in Lean (see `axis_framework.lean`).

2.7.1 Core Results

Definition 2.30 (Axis Orthogonality). Axes A_1, A_2 are *orthogonal* iff neither is derivable from the other: there exist queries answerable by A_1 but not A_2, and vice versa.

Theorem 2.31 (Minimality Implies Orthogonality). If axis set $\mathcal{A}$ is minimal complete for domain D, then all axes in $\mathcal{A}$ are pairwise orthogonal.

Proof. Suppose $\mathcal{A}$ is minimal but contains non-orthogonal axes a, b where a is derivable from b. Then $\mathcal{A} \setminus \{a\}$ remains complete (any query requiring a can be answered via b). This contradicts minimality. ■

Theorem 2.32 (Orthogonality Implies Exchange Property). If $\mathcal{A}$ is an orthogonal axis set, then for any independent subsets $I, J \subseteq \mathcal{A}$ with $|I| < |J|$, there exists $x \in J \setminus I$ such that $I \cup \{x\}$ is independent.

This is the *matroid exchange axiom.* Orthogonal axis sets satisfy it.

Theorem 2.33 (Basis Equicardinality). All minimal complete axis sets for a domain D have the same cardinality.

Proof. Standard matroid theory. If two maximal independent sets had different sizes, the exchange property would yield a contradiction. ■

2.7.2 Why Matroids Matter

The matroid structure has three critical consequences:

1. **Dimension is well-defined.** The "dimension" of a domain (number of required axes) is unique. There are no alternative minimal decompositions with different numbers of axes.
2. **Orthogonality is necessary, not optional.** A type system with redundant (non-orthogonal) axes is provably suboptimal—it contains derivable structure that adds complexity without capability.

3. **The (B, S) decomposition is canonical for standard class semantics.** The two-axis model is not a modeling choice—it is the unique minimal complete axis set for the domain of standard class queries (provenance, identity, compatibility). Extended domains requiring additional queries (e.g., scope-chain resolution) require additional axes (e.g., H for hierarchy). This is exactly what the theory predicts: axes are derived from domain requirements.

Theorem 2.34 (Semantic Non-Embeddability). Let D_1 and D_2 be typing disciplines with axis sets $\mathcal{A}_1 \subset \mathcal{A}_2$ (strict subset). Then:

$$\text{There exists no function } f : D_2 \to D_1 \text{ that preserves query answers.}$$

In particular, nominal typing ($\{B, S\}$) cannot be embedded into structural typing ($\{S\}$).

Proof. Let $a \in \mathcal{A}_2 \setminus \mathcal{A}_1$. By axis independence, $\exists$ query q requiring a. Any embedding f must map types to D_1's representation. But D_1 lacks axis a, so q is unanswerable in D_1. Therefore f cannot preserve q's answers. ∎

Significance: This is an **impossibility theorem about information**, not about implementations. Structural typing cannot "simulate" nominal typing because the information content is strictly smaller. The gap is mathematical, not technical.

---

3 Axis Inclusion Requirements and Fixed-Axis Incompleteness

Framing note: This section establishes dominance relations *within* fixed-axis systems. Section 3.6 proves the stronger result: any fixed-axis system is incomplete for some domain. The correct approach is axis-parametric (deriving the minimal axis set from requirements).

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

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

```
# 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 Static Type System). TypeScript's *static* type system is structural. Class compatibility is determined by shape, not inheritance. However, at runtime, JavaScript's prototype chain provides nominal identity (`instanceof` checks the chain) [25]. This creates a coherence tension discussed in Section 8.7.

The Critical Observation (Semantic Axes):

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

The syntactic name (`__name__`) is metadata in both cases—it doesn't affect which typing discipline is correct. See Definition 2.7 for the distinction between syntactic names and semantic identity.

Binder vs. Observable Identity. In *pure* structural typing, "name" is only a binder/alias for a shape; it is not an observable discriminator. Conformance depends solely on namespace (structure). Any observable discriminator (brand/tag/nominal identity) is an added axis: once it is observable to the conformance relation, the discipline is no longer purely structural.

Lineage axis = ordered identities. The Bases axis B can be viewed as the ordered lineage $\mathrm{MRO}(T)$ (C3 linearization). The "identity" capability is a projection of this lineage: $\mathrm{head}(\mathrm{MRO}(T)) = T$ (exact type), and instance-of is membership $U \in \mathrm{MRO}(T)$. Provenance and conflict resolution are the other projections. There is no separate "I axis"; identity is one of the queries made available by B. A discipline that can only test $\mathrm{head}(\mathrm{MRO}(T))$ has tag identity but not inheritance capabilities; it is a strict subset of full B.

Theorem 3.4 (Axis Inclusion Dominance). When a `bases` axis exists in the class system ($B \neq \emptyset$), systems including $\{B, S\}$ Pareto-dominate all systems using only $\{S\}$ (provides strictly more capabilities with zero additional cost). This dominance is universal, not limited to greenfield development.

Remark: In type system terminology, $\{B, S\}$-inclusive systems are called "nominal typing"; $\{S\}$-only systems are called "structural typing." The theorem establishes that axis inclusion is determined by domain requirements, not preference.

Proof. We prove this in two steps: (1) strict dominance holds unconditionally, (2) retrofit constraints do not constitute an exception.

Step 1: Strict Dominance is Unconditional.

Let $D_{\{S\}}$ be any system using only the $\{S\}$ axis (shape-based). Let $D_{\{B,S\}}$ be a system using both axes.

By Theorem 2.15 (Axis Lattice Dominance):

$$\mathrm{capabilities}(D_{\{S\}}) \subseteq \mathrm{capabilities}(D_{\{B,S\}})$$

By Theorem 2.17 (Capability Completeness), $D_{\{B,S\}}$ provides four capabilities that $D_{\{S\}}$ cannot: provenance, identity, enumeration, conflict resolution.

Therefore: $\mathrm{capabilities}(D_{\{S\}}) \subset \mathrm{capabilities}(D_{\{B,S\}})$ (strict subset).

This dominance holds **regardless of whether the system currently uses these capabilities**. The capability gap exists by the structure of axis subsets, not by application requirements.

Step 2: Retrofit Constraints Do Not Constitute an Exception.

One might object: "In retrofit contexts, external types cannot be made to inherit from my ABCs, so nominal typing is unavailable."

This objection was addressed in Theorem 2.10j (Protocol Dominated by Adapters): when $B \neq \emptyset$, nominal typing with adapters provides all capabilities of Protocol plus four additional capabilities. The "retrofit exception" is not an exception. Adapters are the mechanism that makes nominal typing universally available.

- External type cannot inherit from your ABC? Wrap it in an adapter that does.
- Protocol avoids the adapter? Yes, but avoiding adapters is a convenience, not a capability (Corollary 2.10k).

Conclusion: Systems Excluding Required Axes Have Negative Expected Value.

Given two available options A and B where $\mathrm{capabilities}(A) \subset \mathrm{capabilities}(B)$ and $\mathrm{cost}(A) = \mathrm{cost}(B)$, choosing A is **Pareto-dominated**: there exists no rational justification for A over B under expected utility maximization.

When $B \neq \emptyset$:

- Systems using only $\{S\}$ are Pareto-dominated by systems using $\{B, S\}$
- Same declaration cost: one class definition per interface for both (Definition 2.14c)
- Foreclosure is permanent: Missing capabilities cannot be added later (Theorem 3.67)
- Ignorant choice has expected cost: $E[\mathrm{gap}] > 0$ (Theorem 3.68)
- Retrofit cost dominates analysis cost: $\mathrm{cost}_{\mathrm{retrofit}} > \mathrm{cost}_{\mathrm{analysis}}$ (Theorem 3.69)
- Analysis has positive expected value: $E[V_{\mathrm{analysis}}] > 0$ (Theorem 3.70)

Therefore: **Excluding the B axis when inheritance exists ($B \neq \emptyset$) has negative expected value under capability-based utility.**

Note: In type system terminology, "nominal typing" denotes $\{B, S\}$-inclusive systems; "structural typing" denotes $\{S\}$-only systems.

Note on "what if I don't need the extra capabilities?"

This objection misunderstands option value. A Pareto-dominated choice has negative expected value **even if the additional capabilities are never exercised**:

1. Capability availability has zero marginal cost (identical `isinstance()` syntax)
2. Future requirements are uncertain; capability foreclosure has negative option value (Theorem 3.70)
3. Capability gaps are irreversible: cannot transition from shape to nominal without discipline rewrite (Theorem 3.67)
4. Architecture choices have persistent effects; one-time decisions determine long-term capability sets

The `bases` axis creates an information asymmetry: nominal typing can access inheritance structure; shape-based typing cannot. Adapters ensure nominal typing is universally available.

Theorem 3.71 (Unique Optimum): Under capability-based utility maximization, nominal typing is the unique optimal choice when $B \neq \emptyset$. Choosing shape-based typing while maximizing capabilities contradicts the stated objective (Theorem 3.72: proven incoherence). ■

Theorem 3.5 (Axis Inclusion Dominance, Concrete). Systems including $\{B, S\}$ strictly dominate systems using only $\{S\}$ whenever $B \neq \emptyset$: $\{B, S\}$ systems provide all capabilities of $\{S\}$-only systems plus additional capabilities, at equal or lower cost.

Remark: This theorem is traditionally stated as "nominal typing strictly dominates structural typing." We state it in axis terms to emphasize that the result follows from axis inclusion, not from advocacy for a particular discipline name.

Proof. Consider Python's concrete implementations: - $\{S\}$-only (coherent): `typing.Protocol` (structural typing) - $\{B, S\}$:Abstract Base Classes (ABCs, nominal typing)

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

What Protocols ($\{S\}$-only) 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 ($\{B, S\}$) 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 ⊂ N (strict subset). Both require explicit type declarations. The declaration cost is equivalent: one class definition per interface. Therefore, $\{B, S\}$ systems provide strictly more capabilities at equal or lower cost (earlier failure). ■

Corollary 3.6 (Axis Exclusion Incorrectness). When $B \neq \emptyset$, excluding B from the axis set is not suboptimal: it is incorrect.

Proof. By Theorem 3.5, systems including $\{B, S\}$ strictly dominate systems using only $\{S\}$. By Theorem 2.10j, adapters make $\{B, S\}$ systems universally available. Choosing a strictly dominated option when the superior option is available is definitionally incorrect. ■

Remark: In type system terminology, this means structural typing is incorrect when inheritance exists; nominal typing is correct.

3.1 The Absolute Claim

Claim (Axis Inclusion Correctness). In any programming language with explicit inheritance syntax ($B \neq \emptyset$), systems excluding the B axis (structural typing, Protocol-based typing) are **incorrect**. Duck typing is **incoherent** (Theorem 2.10d). It is not even a valid typing discipline. Systems including $\{B, S\}$

(nominal typing) are **correct**. This is not a preference, recommendation, or tradeoff. It is a mathematical fact derivable from the structure of class systems.

Proof. By Theorem 2.18 (Strict Dominance), systems using $\{B, S\}$ provide all capabilities of $\{S\}$-only systems plus additional capabilities (provenance, type identity, subtype enumeration, type-as-key). By Theorem 2.10j, adapters eliminate the retrofit exception. Therefore, choosing $\{S\}$-only when $B \neq \emptyset$ is choosing the strictly dominated option. ■

What "incorrect" means: 1. **Information-theoretic**: Excluding the B axis discards available information. Discarding available information without compensating benefit is suboptimal by definition. 2. **Capability-theoretic**: Excluding B forecloses capabilities that including it 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.

3.2 Information-Theoretic Foundations

This section establishes the formal foundation of our results. We prove three theorems that make claims about all possible typing disciplines, not just our particular model.

3.8.1 The Impossibility Theorem **Definition 3.10 (Typing Discipline).** A *typing discipline* $\mathcal{D}$ over axis set $A \subseteq \{B, S\}$ is a collection of computable functions that take as input only the projections of types onto axes in A.

Definition 3.11 (Axis-Exclusive Discipline). A $\{S\}$*-only discipline* is a typing discipline over $\{S\}$. It has access to namespaces (shape), but not to the Bases axis.

Remark: In type system terminology, $\{S\}$-only disciplines include structural typing (with declared interfaces) and duck typing (without declared interfaces). Both exclude the B axis.

Remark (Syntactic Names and Type Identity). A discipline that also inspects type identity (e.g., `type(x) is T`) is using information from B, not a third axis. The type identity is $\text{head}(\text{MRO}(T)) = T$, a projection of B that discards all ancestors. Such a discipline has access to $\{S\}$ plus the *leaf* of B, but not the lineage. Since provenance requires ancestor information (which ancestor provides attribute a?), projecting B to its leaf loses exactly the information needed. This is why the impossibility results hold even for disciplines that inspect type identity. Note: the syntactic name (`__name__`) is distinct from type identity (see Definition 2.7).

Definition 3.12 (Provenance Function). The *provenance function* is:

$$\text{prov} : \text{Type} \times \text{Attr} \to \text{Type}$$

where $\text{prov}(T, a)$ returns the type in T's MRO that provides attribute a.

Theorem 3.13 (Provenance Impossibility, Universal). Let $\mathcal{D}$ be ANY $\{S\}$-only discipline (typing discipline over $\{S\}$, even with the leaf of B). Then $\mathcal{D}$ cannot compute prov.

Remark: In type system terminology, this means neither structural typing nor duck typing can provide provenance. Only systems including B (nominal typing) can answer provenance queries.

Proof. We prove this by showing that prov requires information that is information-theoretically absent from (S).

1. **Information content of** (S)**.** A $\{S\}$-only discipline receives: the type name $N(T)$ and the namespace $S(T) = \{a_1, a_2, \ldots, a_k\}$ (the set of attributes T declares or inherits).

2. **Information content required by prov.** The function $\text{prov}(T, a)$ must return *which ancestor type* originally declared a. This requires knowing the MRO of T and which position in the MRO declares a.

3. **MRO is defined exclusively by** B**.** By Definition 2.11, $\text{MRO}(T) = \text{C3}(T, B(T))$. The C3 linearization of T's base classes. The function $B : \text{Type} \to \text{List[Type]}$ is the Bases axis.

4. (S) **contains no information about** B**.** The namespace $S(T)$ is the *union* of attributes from all ancestors. It does not record *which* ancestor contributed each attribute. Two types with identical S can have completely different B (and therefore different MROs and different provenance answers).

5. **Concrete counterexample.** Let:

   - $A = \text{type}(\text{"A"}, (), \{\text{"x"} : 1\})$
   - $B_1 = \text{type}(\text{"B1"}, (A,), \{\})$
   - $B_2 = \text{type}(\text{"B2"}, (), \{\text{"x"} : 1\})$

   Then $S(B_1) = S(B_2) = \{\text{"x"}\}$ (both have attribute "x"), but:

   - $\text{prov}(B_1, \text{"x"}) = A$ (inherited from parent)
   - $\text{prov}(B_2, \text{"x"}) = B_2$ (declared locally)

   A $\{S\}$-only discipline cannot distinguish B_1 from B_2, therefore cannot compute prov. ■

Corollary 3.14 (No Algorithm Exists). There exists no algorithm, heuristic, or approximation that allows a $\{S\}$-only discipline to compute provenance. This is not a limitation of current implementations: it is information-theoretically impossible.

Proof. The proof of Theorem 3.13 shows that the input (S) contains strictly less information than required to determine prov. No computation can extract information that is not present in its input. ■

Significance: This is not "our model doesn't have provenance." It is "NO model over (S) can have provenance." The impossibility is mathematical, not implementational.

Theorem 3.13a (Semantic Non-Embeddability). Nominal typing cannot be embedded into structural typing. Formally: there exists no function $f : \mathcal{D}_{\{B,S\}} \to \mathcal{D}_{\{S\}}$ that preserves type query answers.

Proof. By Theorem 3.13, provenance is answerable in $\mathcal{D}_{\{B,S\}}$ but not in $\mathcal{D}_{\{S\}}$. Any embedding must preserve query answers. Since no function in $\mathcal{D}_{\{S\}}$ computes provenance, no such embedding exists. ■

Corollary 3.13b (Strict Information Gap). The gap between nominal and structural typing is not a matter of convenience or expressiveness—it is an information-theoretic impossibility. Structural typing has strictly less information content than nominal typing. No encoding, simulation, or workaround can bridge this gap.

This result generalizes: for any axis sets $\mathcal{A}_1 \subset \mathcal{A}_2$, the discipline using $\mathcal{A}_1$ cannot embed the discipline using $\mathcal{A}_2$.

3.8.2 The Derived Characterization Theorem A potential objection is that our capability enumeration $\mathcal{C}_B = \{\text{provenance, identity, enumeration, conflict resolution}\}$ is arbitrary. We now prove it is **derived from information structure**, not chosen.

Definition 3.15 (Query). A *query* is a computable function $q : \text{Type}^k \to \text{Result}$ that a typing discipline evaluates.

Definition 3.16 (Shape-Respecting Query). A query q is *shape-respecting* if for all types with $S(A) = S(B)$:

$$q(\ldots, A, \ldots) = q(\ldots, B, \ldots)$$

That is, shape-equivalent types produce identical query results.

Definition 3.17 (B-Dependent Query). A query q is *B-dependent* if there exist types A, B with $S(A) = S(B)$ but $q(A) \neq q(B)$.

Theorem 3.18 (Query Space Partition). Every query is either shape-respecting or B-dependent. These categories are mutually exclusive and exhaustive.

Proof. - *Mutual exclusion:* If q is shape-respecting, then $S(A) = S(B) \Rightarrow q(A) = q(B)$. If q is B-dependent, then $\exists A, B : S(A) = S(B) \land q(A) \neq q(B)$. These are logical negations. - *Exhaustiveness:* For any query q, either $\forall A, B : S(A) = S(B) \Rightarrow q(A) = q(B)$ (shape-respecting) or $\exists A, B : S(A) = S(B) \land q(A) \neq q(B)$ (B-dependent). Tertium non datur. ■

Theorem 3.19 (Capability Gap = B-Dependent Queries). The capability gap between systems using only $\{S\}$ and systems using $\{B, S\}$ is EXACTLY the set of B-dependent queries:

$$\text{Capabilities}_{\{B,S\}} \setminus \text{Capabilities}_{\{S\}} = \{q : q \text{ is B-dependent}\}$$

Remark: In type system terminology, this characterizes the gap between structural/duck typing ($\{S\}$-only) and nominal typing ($\{B, S\}$).

Proof. - ($\supseteq$) If q is B-dependent, then $\exists A, B$ with $S(A) = S(B)$ but $q(A) \neq q(B)$. Shape disciplines cannot distinguish A from B, so cannot compute q. Nominal disciplines have access to B, so can distinguish A from B via MRO. Therefore q is in the gap. - ($\subseteq$) If q is in the gap, then nominal can compute it but shape cannot. If q were shape-respecting, shape could compute it (contradiction). Therefore q is B-dependent. ■

Theorem 3.20 (Four Capabilities Are Complete). The set $\mathcal{C}_B = \{\text{provenance, identity, enumeration, conflict resolution}\}$ is the complete set of B-dependent query classes.

Proof. We show that every B-dependent query reduces to one of these four:

1. **Provenance queries** ("which type provided a?"): Any query requiring ancestor attribution.
2. **Identity queries** ("is x an instance of T?"): Any query requiring MRO membership.
3. **Enumeration queries** ("what are all subtypes of T?"): Any query requiring inverse MRO.
4. **Conflict resolution queries** ("which definition wins?"): Any query requiring MRO ordering.

Completeness argument: A B-dependent query must use information from B. The only information in B is: - Which types are ancestors (enables identity, provenance) - The order of ancestors (enables conflict resolution) - The inverse relation (enables enumeration)

These three pieces of information (ancestor set, ancestor order, inverse relation) generate exactly four query classes. No other information exists in B. ■

Corollary 3.21 (Capability Set Is Minimal). $|\mathcal{C}_B| = 4$ and no element is redundant.

Proof. Each capability addresses a distinct aspect of B: - Provenance: forward lookup by attribute - Identity: forward lookup by type - Enumeration: inverse lookup - Conflict resolution: ordering

Removing any one leaves queries that the remaining three cannot answer. ■

3.8.3 The Axis Packing Dichotomy

A common objection to the capability gap is "packing": encoding B into S to recover B-dependent queries without adding an axis. We prove this is impossible.

Definition 3.22a (Full Semantics). The full semantics of a type is:

$$\text{Full} := S \times B$$

where S is the namespace (shape) and B is the bases (inheritance).

Definition 3.22b (Shape Observation). The shape observation function is the projection:

$$\text{obs}_S : \text{Full} \to S, \quad \text{obs}_S(s, b) = s$$

This formalizes what structural/shape-based typing can see: only the namespace.

Definition 3.22c (Shape-Only Definability). A query $q : \text{Full} \to Y$ is *shape-only* iff it factors through obs_S:

$$\exists f : S \to Y.\ q = f \circ \text{obs}_S$$

That is, $q(s, b) = f(s)$ for all $(s, b) \in \text{Full}$.

Definition 3.22d (B-Dependence). A query $q : \text{Full} \to Y$ *depends on* B iff:

$$\exists s, b_1, b_2.\ b_1 \neq b_2 \wedge q(s, b_1) \neq q(s, b_2)$$

That is, q distinguishes types with identical shape but different bases.

Theorem 3.22e (No Free Lunch for Axis Packing). If q depends on B, then q is not shape-only.

Proof. Suppose $q = f \circ \text{obs}_S$ for some f. Let s, b_1, b_2 witness B-dependence. Then:

$$q(s, b_1) = f(\text{obs}_S(s, b_1)) = f(s) = f(\text{obs}_S(s, b_2)) = q(s, b_2)$$

But $q(s, b_1) \neq q(s, b_2)$ by B-dependence. Contradiction. ■

This is the core impossibility: if the observation interface is only S, then no query that distinguishes on B can be defined.

Definition 3.22f (Packing). A *packing* of B into an extended shape S' is an encoding:

$$\text{enc} : \text{Full} \to S'$$

plus an extended observation:

$$\text{obs}_{S'} : S' \to S$$

such that the shape interface is preserved:

$$\text{obs}_{S'}(\text{enc}(s, b)) = s \quad \text{for all } (s, b)$$

This formalizes "packing B into S without changing what structural typing observes."

Theorem 3.22g (Packing Cannot Recover B-Queries). Let enc, $\text{obs}_{S'}$ be a packing that preserves the S-interface. Let $q_B : \text{Full} \to Y$ be any B-dependent query. Then there is no $q' : S' \to Y$ that is shape-only w.r.t. $\text{obs}_{S'}$ and satisfies:

$$q'(\text{enc}(s, b)) = q_B(s, b) \quad \text{for all } (s, b)$$

Proof. Suppose $q' = f \circ \text{obs}_{S'}$ for some $f : S \to Y$. Let s, b_1, b_2 witness B-dependence of q_B. Then:

$$q_B(s, b_1) = q'(\text{enc}(s, b_1)) = f(\text{obs}_{S'}(\text{enc}(s, b_1))) = f(s)$$
$$q_B(s, b_2) = q'(\text{enc}(s, b_2)) = f(\text{obs}_{S'}(\text{enc}(s, b_2))) = f(s)$$

Therefore $q_B(s, b_1) = q_B(s, b_2)$, contradicting B-dependence. ■

Theorem 3.22h (The Axis Packing Dichotomy). To make B-dependent queries answerable after packing, **exactly one** of the following must happen:

1. **Extend the observable interface.** Introduce an operation $\pi_B : S' \to B$ (or equivalent) that exposes B information. This is *adding an axis*, so B becomes observable.
2. **Fail to answer required queries.** Remain shape-only w.r.t. $\text{obs}_{S'}$ and lose provenance, identity, enumeration, and conflict resolution.

Proof. By Theorem 3.22g, if the S-interface is preserved, B-dependent queries are unanswerable. To answer them, the interface must be extended. Adding π_B makes B observable. This is precisely what "adding an axis" means: making new information available to queries. ■

Corollary 3.22i (Packing Maneuver Is Futile). The "packing" objection to the capability gap fails in all cases:

- If packing preserves the S-interface: B-queries remain impossible (Theorem 3.22g).
- If packing extends the interface with π_B: this is adding an axis, not avoiding one.

There is no third option. "Packing B into S" is revealed as either useless (information hidden) or equivalent to adding an axis (information exposed).

Remark 3.22j (Categorical Interpretation). The dichotomy has a clean categorical form: given the product $\text{Full} = S \times B$ with canonical projections π_S, π_B, any query depending on π_B must factor through π_B. Packing replaces the product with an encoding S' that generally lacks canonical projections. To recover B-queries, one must add $\pi_B : S' \to B$ as an explicit operation—i.e., restore the axis. The product structure is not an accident; it is the minimal representation preserving all queryable information.

3.8.4 The Complexity Lower Bound Theorem Our O(1) vs $\Omega(n)$ complexity claim requires proving that $\Omega(n)$ is a **lower bound**, not merely an upper bound. We must show that NO algorithm can do better.

Definition 3.22 (Computational Model). We formalize error localization as a decision problem in the following model:

- **Input:** A program P with n call sites $c_1, \ldots, c_n$, each potentially accessing attribute a on objects of type T.
- **Oracle:** The algorithm may query an oracle $\mathcal{O}(c_i) \in \{\text{uses } a, \text{does not use } a\}$ for each call site.
- **Output:** The set $V \subseteq \{c_1, \ldots, c_n\}$ of call sites that access a on objects lacking a.

- **Correctness:** The algorithm must output the exact set V for all valid inputs.

This model captures duck typing's fundamental constraint: type compatibility is checked at each call site, not at declaration.

Definition 3.23 (Inspection Cost). The *cost* of an algorithm is the number of oracle queries in the worst case over all inputs.

Theorem 3.24 (Duck Typing Lower Bound). Any algorithm that correctly solves error localization in the above model requires $\Omega(n)$ oracle queries in the worst case.

Proof. By adversary argument and information-theoretic counting.

1. **Adversary construction.** Fix any deterministic algorithm $\mathcal{A}$. We construct an adversary that forces $\mathcal{A}$ to query at least $n-1$ call sites.
2. **Adversary strategy.** The adversary maintains a set S of "candidate violators", call sites that could be the unique violating site. Initially $S = \{c_1, \ldots, c_n\}$. When $\mathcal{A}$ queries $\mathcal{O}(c_i)$:
   - If $|S| > 1$: Answer "does not use a" and set $S \leftarrow S \setminus \{c_i\}$
   - If $|S| = 1$: Answer consistently with $c_i \in S$ or $c_i \notin S$
3. **Lower bound derivation.** The algorithm must distinguish between n possible inputs (exactly one of $c_1, \ldots, c_n$ violates). Each query eliminates at most one candidate. After $k < n-1$ queries, $|S| \geq 2$, so the algorithm cannot determine the unique violator. Therefore $\mathcal{A}$ requires at least $n-1 \in \Omega(n)$ queries.
4. **Generalization.** For the case where multiple call sites may violate: there are 2^n possible subsets. Each binary query provides at most 1 bit. Therefore $\log_2(2^n) = n$ queries are necessary to identify the exact subset. ■

Remark (Static Analysis). Static analyzers precompute call site information via control-flow analysis over the program text. This shifts the $\Omega(n)$ cost to analysis time rather than eliminating it. The bound characterizes the inherent information content required, n bits to identify n potential violation sites, regardless of when that information is gathered.

Theorem 3.25 (Nominal Typing Upper Bound). Nominal error localization requires exactly 1 inspection.

Proof. In nominal typing, constraints are declared at the class definition. The constraint "type T must have attribute a" is checked at the single location where T is defined. If the constraint is violated, the error is at that location. No call site inspection is required. ■

Corollary 3.26 (Complexity Gap Is Unbounded). The ratio $\frac{\text{DuckCost}(n)}{\text{NominalCost}}$ grows without bound:

$$\lim_{n \to \infty} \frac{\Omega(n)}{O(1)} = \infty$$

Proof. Immediate from Theorems 3.24 and 3.25. ■

Corollary 3.27 (Lower Bound Is Tight). The $\Omega(n)$ lower bound for duck typing is achieved by naive inspection; no algorithm can do better, and simple algorithms achieve this bound.

Proof. Theorem 3.24 proves $\Omega(n)$ is necessary. Linear scan of call sites achieves $O(n)$. Therefore the bound is tight. ■

3.3 Summary: Core Formal Results

We have established four theorems with universal scope:

Theorem	Statement	Proof Technique
3.13 (Impossibility)	No shape discipline can compute provenance	Information-theoretic: input lacks required data
3.19 (Derived Characterization)	Capability gap = B-dependent queries	Mathematical: query space partitions exactly
3.22h (Packing Dichotomy)	Packing B into S either fails or adds an axis	Categorical: observation preservation blocks recovery
3.24 (Lower Bound)	Duck typing requires $\Omega(\mathrm{n})$ inspections	Adversary argument: any algorithm can be forced

These are not claims about our model; they are claims about **the universe of possible typing systems**. The theorems establish:

- Theorem 3.13 proves no model over (S) can provide provenance.
- Theorem 3.19 proves the capability enumeration is derived from information structure.
- Theorem 3.22h proves "packing" cannot circumvent the capability gap; it either fails or is equivalent to adding an axis.
- Theorem 3.24 proves no algorithm can overcome the information-theoretic limitation.

These results follow from the structure of the problem, not from our particular formalization.

3.4 Information-Theoretic Completeness

For completeness, we restate the original characterization in the context of the new foundations.

Definition 3.28 (Query). A *query* is a predicate $q : \mathrm{Type} \to \mathrm{Bool}$ that a typing discipline can evaluate.

Definition 3.29 (Axis-Respecting Query). A query q is $\{S\}$*-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 3.30 (Capability Gap Characterization). Let $\{S\}$-Queries be the set of all $\{S\}$-respecting queries, and let AllQueries be the set of all queries. If there exist types $A \neq B$ with $S(A) = S(B)$, then:

$$\{S\}\text{-Queries} \subsetneq \text{AllQueries}$$

Remark: In type system terminology, "$\{S\}$-Queries" corresponds to structural/shape/duck typing capabilities.

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

Corollary 3.31 (Derived Capability Set). The capability gap between $\{S\}$-only and $\{B, S\}$ systems is **exactly** the set of queries that depend on the Bases axis:

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

Remark: This characterizes the gap between structural/duck typing ($\{S\}$) and nominal typing ($\{B, S\}$).

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 removes the ability to answer 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) cannot be answered. This follows from the structure of the information available.

3.5 Completeness and Robustness Theorems

This section presents additional theorems that establish the completeness and robustness of our results. Each theorem addresses a specific aspect of the model's coverage.

3.11.1 Model Completeness **Theorem 3.32 (Model Completeness).** The (B, S) model captures all information constitutive of a type. Any computable function over types is expressible as a function of (B, S).

Remark (Two Senses of Completeness). "Model completeness" (this theorem) means (B, S) captures all constitutive information of a type object. "Axis set completeness for domain D" (Definition 2.22) means the axis set can answer all queries in D. These are distinct: (B, S) is model-complete (captures all type information) but may be domain-incomplete (cannot answer queries requiring axes beyond B and S, such as scope-chain queries requiring H).

Proof. The proof proceeds by constitutive definition, not empirical enumeration.

In Python, `type(name, bases, namespace)` is the universal type constructor. Every type T is created by some invocation `type(n, b, s)`, either explicitly or via the `class` statement (which is syntactic sugar for `type()`). A type does not merely *have* (B, S); a type *is* (B, S). There is no other information constitutive of a type object.

Therefore, for any computable function $g : \text{Type} \to \alpha$:

$$g(T) = g(\texttt{type}(n, b, s)) = h(n, b, s)$$

for some computable h. Any function of a type is definitionally a function of the triple that constitutes it.

Remark (Derived vs. Constitutive). Properties like `__mro__` (method resolution order) or `__module__` are not counterexamples: MRO is computed from B by C3 linearization; `__module__` is stored in the namespace S. These are *derived from* or *contained in* (B, S), not independent of it.

This is a definitional closure: a critic cannot exhibit a third axis because any proposed axis is either (a) stored in S (including `__name__`), (b) computable from (B, S), or (c) not part of the type's semantic identity (e.g., memory address). ■

Corollary 3.33 (No Hidden Information). There exists no third axis that shape-based typing could use to recover provenance. The information is structurally absent, not because we failed to model it, but because types *are* (B, S) by construction.

3.11.2 Capability Comparison **Theorem 3.34 (Capability Superset).** Let $\mathcal{C}_{\text{duck}}$ be the capabilities available under duck typing. Let $\mathcal{C}_{\text{nom}}$ be the capabilities under nominal typing. Then:

$$\mathcal{C}_{\text{duck}} \subseteq \mathcal{C}_{\text{nom}}$$

Proof. Duck typing operations are: 1. Attribute access: `getattr(obj, "name")` 2. Attribute existence: `hasattr(obj, "name")` 3. Method invocation: `obj.method()`

All three operations are available in nominal systems. Nominal typing adds type identity operations; it does not remove duck typing operations. ■

Theorem 3.35 (Strict Superset). The inclusion is strict:

$$\mathcal{C}_{\text{duck}} \subsetneq \mathcal{C}_{\text{nom}}$$

Proof. Nominal typing provides provenance, identity, enumeration, and conflict resolution (Theorem 2.17). Duck typing cannot provide these (Theorem 3.13). Therefore:

$$\mathcal{C}_{\text{nom}} = \mathcal{C}_{\text{duck}} \cup \mathcal{C}_B$$

where $\mathcal{C}_B \neq \emptyset$. ■

Corollary 3.36 (No Capability Tradeoff). Choosing nominal typing over duck typing: - Forecloses **zero** capabilities - Gains **four** capabilities

There is no capability tradeoff. Nominal typing strictly dominates.

Remark (Capability vs. Code Compatibility). The capability superset does not mean "all duck-typed code runs unchanged under nominal typing." It means "every operation expressible in duck typing is expressible in nominal typing." The critical distinction:

- **False equivalence** (duck typing): `WellFilterConfig` and `StepWellFilterConfig` are structurally identical but semantically distinct (different MRO positions, different scopes). Duck typing conflates them; it literally cannot answer "which type is this?" This is not flexibility; it is **information destruction**.
- **Type distinction** ($\{B, S\}$ systems): `isinstance(config, StepWellFilterConfig)` distinguishes them in O(1). The distinction is expressible because $\{B, S\}$ systems preserve type identity.

The "acceptance" of structurally-equivalent types in $\{S\}$-only systems is not a capability; it is the *absence* of the capability to distinguish them. $\{B, S\}$ systems add this capability without removing any structural typing operation. See Case Study 1 (§5.2, Theorem 5.1) for the complete production example demonstrating that structural identity $\neq$ semantic identity.

Remark: The traditional terms "duck typing" and "nominal typing" correspond to $\{S\}$-only and $\{B, S\}$ systems respectively.

3.11.3 Axiom Justification

Lemma 3.37 (Axis-Exclusion is Definitional). The axiom "$\{S\}$-only typing treats same-namespace types identically" is not an assumption; it is the **definition** of $\{S\}$-only typing.

Proof. $\{S\}$-only typing is defined as a typing discipline over $\{S\}$ only (Definition 2.10). If a discipline uses information from B (the Bases axis) to distinguish types, it is, by definition, not $\{S\}$-only.

The axiom is not: "We assume $\{S\}$-only typing can't distinguish same-shape types." The axiom is: "$\{S\}$-only typing means treating same-shape types identically."

Any system that distinguishes same-shape types is using B (explicitly or implicitly). ■

Remark: In type system terminology, "$\{S\}$-only typing" corresponds to shape-based/duck/structural typing.

Corollary 3.38 (No Axis-Extension Without Including Axis). There exists no "clever" $\{S\}$-only system that can distinguish types A and B with $S(A) = S(B)$. Such a system would, by definition, not be $\{S\}$-only.

3.11.4 Extension Impossibility

Theorem 3.39 (Extension Impossibility). Let $\mathcal{D}$ be any $\{S\}$-only typing system. Let $\mathcal{D}'$ be $\mathcal{D}$ extended with any computable function $f : \text{Namespace} \to \alpha$. Then $\mathcal{D}'$ still cannot compute provenance.

Proof. Provenance requires distinguishing types A and B where $S(A) = S(B)$ but $\text{prov}(A, a) \neq \text{prov}(B, a)$ for some attribute a.

Any function $f : \text{Namespace} \to \alpha$ maps A and B to the same value, since $S(A) = S(B)$ implies f receives identical input for both.

Therefore, f provides no distinguishing information. The only way to distinguish A from B is to use information not in Namespace, i.e., the Bases axis B.

No computable extension over $\{S\}$ (even with the leaf of B) can recover provenance. ■

Corollary 3.40 (No Future Fix). No future language feature, library, or tool operating within the duck typing paradigm can provide provenance. The limitation is structural, not technical.

3.11.5 Capability Exhaustiveness

Theorem 3.43a (Capability Exhaustiveness). The four capabilities (provenance, identity, enumeration, conflict resolution) are **exhaustive**; they are the only capabilities derivable from the Bases axis.

Proof. (Machine-checked in `nominal_resolution.lean`, Section 6: CapabilityExhaustiveness)

The Bases axis provides MRO, a **list of types**. A list has exactly three queryable properties: 1. **Ordering**: Which element precedes which? $\to$ *Conflict resolution* (C3 linearization selects based on MRO order) 2. **Membership**: Is element X in the list? $\to$ *Enumeration* (subtype iff in some type's MRO) 3. **Element identity**: Which specific element? $\to$ *Provenance* and *type identity* (distinguish structurally-equivalent types by MRO position)

These are exhaustive by the structure of lists; there are no other operations on a list that do not reduce to ordering, membership, or element identity. Therefore, the four capabilities are derived from MRO structure, not enumerated by inspection. ■

Corollary 3.43b (No Missing Capability). Any capability claimed to require B reduces to one of the four. There is no "fifth capability" that B provides.

Proof. Any operation on B is an operation on MRO. Any operation on MRO is an operation on a list. List operations are exhaustively {ordering, membership, identity}. ∎

Theorem 3.43b-bis (Capability Reducibility). Every B-dependent query reduces to a composition of the four primitive capabilities.

Proof. Let $q : \text{Type} \to \alpha$ be any B-dependent query (per Definition 3.17). By Definition 3.17, q distinguishes types with identical structure: $\exists A, B : S(A) = S(B) \wedge q(A) \neq q(B)$.

The only information distinguishing A from B is $B(A) \neq B(B)$ (different bases), which distinguishes via:

- Ancestor membership: is $T \in \text{ancestors}(A)$? $\to$ covered by **provenance**
- Subtype enumeration: what are all $T : T <: A$? $\to$ covered by **enumeration**
- MRO position: which type wins for attribute a? $\to$ covered by **conflict_resolution**
- Type identity: is $\text{head}(\text{MRO}(x)) = T$? $\to$ covered by **type_identity**

Note: Syntactic names (`__name__`) are metadata external to this model (see Definition 2.7).

No other distinguishing information exists (Theorem 3.32: (B, S) is complete).

Therefore any B-dependent query q can be computed by composing:

$$q(T) = f(\text{provenance}(T), \text{identity}(T), \text{enumeration}(T), \text{conflict\_resolution}(T))$$

for some computable f. ∎

3.11.6a Adapter Cost Analysis

Theorem 3.43c (Adapter Declaration is Information-Preserving). An adapter declares information that is **already true**, that a type conforms to an interface. Declaration does not create the conformance; it makes it explicit.

Proof. If `TheirType` does not satisfy `YourABC`'s interface, the adapter fails at definition time (missing method error). If `TheirType` does satisfy the interface, the conformance existed before the adapter. The adapter is not implementation; it is documentation of pre-existing fact. ∎

Theorem 3.43d (Adapter Amortization). Adapter cost is O(1). Manual capability implementation is O(N) where N is the number of use sites.

Proof. (Machine-checked in `nominal_resolution.lean`, Section 7: AdapterAmortization)

Under nominal typing (with adapter): - Provenance: Automatic via `type(obj).__mro__` (0 additional code per use) - Identity: Automatic via `isinstance()` (0 additional code per use) - Enumeration: Automatic via `__subclasses__()` (0 additional code per use) - Conflict resolution: Automatic via C3 (0 additional code per use)

Under structural typing (without adapter), to recover any capability manually: - Provenance: Must thread source information through call sites (1 additional parameter × N calls) - Identity: Must maintain external type registry (1 registry + N registration calls) - Enumeration: Must maintain external subtype set (1 set + N insertions) - Conflict resolution: Must implement manual dispatch (1 dispatcher + N cases)

The adapter is 2 lines. Manual implementation is $\Omega(N)$. For $N \geq 1$, adapter dominates. ∎

Corollary 3.43e (Negative Adapter Cost). Adapter "cost" is negative, a net benefit.

Proof. The adapter enables automatic capabilities that would otherwise require O(N) manual implementation. The adapter costs O(1). For any system requiring the capabilities, adapter provides net savings of $\Omega(N) - O(1) = \Omega(N)$. The "cost" is negative. ∎

Corollary 3.43f (Adapter Cost Objection is Invalid). Objecting to adapter cost is objecting to O(1) overhead while accepting O(N) overhead. This is mathematically incoherent.

3.11.6b Methodological Independence

Theorem 3.43g (Methodological Independence). The dominance theorems are derived from the structure of (B, S), not from any implementation. OpenHCS is an existential witness, not a premise.

Proof. We distinguish two logical roles:

- **Premise:** A proposition the conclusion depends on. If false, the conclusion may not follow.
- **Existential witness:** A concrete example demonstrating satisfiability. Removing it does not affect the theorem's validity.

Examine the proof of Theorem 3.13 (Provenance Impossibility): it shows that (S) contains insufficient information to compute provenance. This is an information-theoretic argument referencing no codebase. The proof could be written before any codebase existed.

Proof chain (no OpenHCS references):

1. Theorem 2.17 (Capability Gap): Proved from the definition of $\{S\}$-only typing
2. Theorem 3.5 (Strict Dominance): Proved from Theorem 2.17 + Theorem 2.18
3. Theorem 2.10j (Adapters): Proved from capability comparison

OpenHCS appears only to demonstrate that the four capabilities are *achievable*, that real systems use provenance, identity, enumeration, and conflict resolution. This is an existence proof ("such systems exist"), not a premise ("if OpenHCS works, then the theorems hold").

Analogy: Proving "comparison-based sorting requires $\Omega(n \log n)$" does not require testing on multiple arrays. Exhibiting quicksort demonstrates achievability, not theorem validity. ■

Corollary 3.43h (Cross-Codebase Validity). The theorems apply to any codebase in any language where $B \neq \emptyset$. OpenHCS is a sufficient example, not a necessary one.

3.11.6c Inheritance Ubiquity

Theorem 3.43i (Inheritance Ubiquity). In Python, $B = \emptyset$ requires actively avoiding all standard tooling. Any project using ≥ 1 of the following has $B \neq \emptyset$ by construction:

Category	Examples	Why $B \neq \emptyset$
Exceptions	`raise MyError()`	Must subclass `Exception`
Web frameworks	Django, Flask, FastAPI	Views/models inherit framework bases
Testing	pytest classes, unittest	Test classes inherit `TestCase` or use class fixtures
ORM	SQLAlchemy, Django ORM	Models inherit declarative `Base`
Data validation	Pydantic, attrs	Models inherit `BaseModel`
Enumerations	`class Color(Enum)`	Must subclass `Enum`
Abstract interfaces	ABC, Protocol with inheritance	Defines inheritance hierarchy
Dataclasses	`@dataclass` with inheritance	Parent class in `__bases__`
Context managers	Class-based `__enter__/__exit__`	Often inherit helper bases
Type extensions	`typing.NamedTuple`, `TypedDict`	Inherit from typing constructs

Proof. Each listed feature requires defining or inheriting from a class with non-trivial bases. In Python, even an "empty" class `class X: pass` has `X.__bases__ == (object,)`, so $B \supseteq \{\texttt{object}\}$. For $B = \emptyset$ to hold, a project must use:

- No user-defined exceptions (use only built-in exceptions)
- No web frameworks (no Django, Flask, FastAPI, Starlette, etc.)
- No ORM (no SQLAlchemy, Django ORM, Peewee, etc.)
- No Pydantic, attrs, or dataclass inheritance
- No Enum
- No ABC or Protocol inheritance
- No pytest/unittest class-based tests
- No class-based context managers
- Pure functional style with only module-level functions and built-in types

This describes a pathologically constrained subset of Python, not "most code" but "no OOP at all." ■

Corollary 3.43j (B=∅ Is Exceptional). The $B = \emptyset$ case applies only to: 1. Languages without inheritance by design (Go) 2. Pure data serialization boundaries (JSON parsing before domain modeling) 3. FFI boundaries (ctypes, CFFI) before wrapping in domain types 4. Purely functional codebases with no class definitions

In all other cases—which constitute the overwhelming majority of production Python, Java, C#, TypeScript, Kotlin, Swift, Scala, and C++ code—$B \neq \emptyset$ and nominal typing strictly dominates.

Corollary 3.43k (Inheritance Prevalence). A claim that "$B = \emptyset$ is the common case" would require exhibiting a non-trivial production codebase using none of the tooling in Theorem 3.43i. No such codebase is known to exist in the Python ecosystem.

3.11.7 Generics and Parametric Polymorphism

Theorem 3.43 (Generics Preserve Axis Structure). Parametric polymorphism does not introduce a third axis. Type parameters refine (B, S), not add orthogonal information.

Proof. A parameterized type $G\langle T\rangle$ (e.g., `List<Dog>`) has:

- $B(G\langle T\rangle) = B(G)[T/\tau]$. Bases with parameter substituted
- $S(G\langle T\rangle) = S(G)[T/\tau]$. Namespace with parameter in signatures

The type parameter affects both B and S after substitution. No additional axis is required; generics are fully captured by (B, S). ■

Theorem 3.44 (Generic Shape Indistinguishability). Under shape-based typing, `List<Dog>` and `Set<Cat>` are indistinguishable if $S(\text{List}\langle\text{Dog}\rangle) = S(\text{Set}\langle\text{Cat}\rangle)$.

Proof. Shape typing uses only S. If two parameterized types have the same method signatures (after parameter substitution), shape typing treats them identically. It cannot distinguish: - The base generic type (`List` vs `Set`) - The type parameter (`Dog` vs `Cat`) - The generic inheritance hierarchy

These require N (for parameter identity) and B (for hierarchy). ■

Theorem 3.45 (Generic Capability Gap Extends). The four capabilities from $\mathcal{C}_B$ (provenance, identity, enumeration, conflict resolution) apply to generic types. Generics do not reduce the capability gap; they **increase the type space** where it applies.

Proof. For generic types, the four capabilities manifest as: 1. **Provenance:** "Which generic type provided this method?" — requires B 2. **Identity:** "Is this `List<Dog>` or `Set<Cat>`?" — requires parameterized N 3. **Enumeration:** "What are the subtypes of `Collection<T>`?" — requires B 4. **Conflict resolution:** "Which `Comparable<T>` implementation wins?" — requires B

Additionally, generics introduce **variance** (covariant, contravariant, invariant), which requires B to track inheritance direction. Shape typing discards B and the parameter component of N, losing all four capabilities plus variance. ■

Corollary 3.45.1 (Same Four, Larger Space). Generics do not create new capabilities—they apply the same four capabilities to a larger type space. The capability gap is preserved, not reduced.

Theorem 3.46 (Erasure Does Not Save Shape Typing). In languages with type erasure (Java), the capability gap still exists [29].

Proof. Type checking occurs at compile time, where full parameterized types are available. Erasure only affects runtime representations. Our theorems about typing disciplines apply to the type system (compile time), not runtime behavior.

At compile time: - The type checker has access to `List<Dog>` vs `List<Cat>` - Shape typing cannot distinguish them if method signatures match - Nominal typing can distinguish them

At runtime (erased): - Both become `List` (erased) - Shape typing cannot distinguish `ArrayList` from `LinkedList` - Nominal typing can (via `instanceof`)

The capability gap exists at both levels. ■

Theorem 3.47 (Universal Extension). All capability gap theorems (3.13, 3.19, 3.24) extend to generic type systems. The formal results apply to:

- **Erased generics:** Java, Scala, Kotlin
- **Reified generics:** C#, Kotlin (inline reified)
- **Monomorphized generics:** Rust, C++ (templates)

- **Compile-time only:** TypeScript, Swift

Proof. Each language encodes generics as parameterized N (see Table 2.2). The (B, S) model applies uniformly. Type checking occurs at compile time where full parameterized types are available. Runtime representation (erased, reified, or monomorphized) is irrelevant to typing discipline. ∎

Corollary 3.48 (No Generic Escape). Generics do not provide an escape from the capability gap. No major language invented a fourth axis.

Remark 3.49 (Exotic Type Features). Intersection types, union types, row polymorphism, higher-kinded types, and multiple dispatch do not escape the (B, S) model:

- **Intersection/union types** (TypeScript `A & B`, `A | B`): Refine N, combine B and S. Still two axes (N derivable from B).
- **Row polymorphism** (OCaml `< x: int; .. >`): Pure structural typing using S only, but with a *declared* interface (unlike duck typing). OCaml row types are coherent (Theorem 2.10d does not apply) because the object types and row variables are declared explicitly [18]; they still lose the four B-dependent capabilities (provenance, identity, enumeration, conflict resolution) and cannot provide metaprogramming hooks (Theorem 2.10p).
- **Higher-kinded types** (Haskell `Functor`, `Monad`): Parameterized N at the type-constructor level. Typeclass hierarchies provide B.
- **Multiple dispatch** (Julia): Type hierarchies exist (`AbstractArray <: Any`). B axis present. Dispatch semantics are orthogonal to type structure.
- **Prototype-based inheritance** (JavaScript): Prototype chain IS the B axis at object level. `Object.getPrototypeOf()` traverses MRO.

No mainstream type system feature introduces a third axis orthogonal to (B, S).

3.11.7 Scope Expansion: From Greenfield to Universal

Theorem 3.50 (Universal Optimality). Wherever inheritance hierarchies exist and are accessible, nominal typing provides strictly more capabilities than shape-based typing. This is not limited to greenfield development.

Proof. The capability gap (Theorem 3.19) is information-theoretic: shape typing discards B, losing four capabilities. This holds regardless of: - Whether code is new or legacy - Whether the language is compiled or interpreted - Whether types are manifest or inferred - Whether the system uses classes, traits, protocols, or typeclasses

The gap exists wherever B exists. ∎

Corollary 3.51 (Scope of Shape Typing). Shape-based typing is capability-equivalent to nominal typing only when:

1. **No hierarchy exists:** $B = \emptyset$ (e.g., Go interfaces, JSON objects)
2. **Hierarchy is inaccessible:** True FFI boundaries where type metadata is lost

When $B \neq \emptyset$, shape-based typing is **always dominated** by nominal typing with adapters (Theorem 2.10j). "Deliberately ignored" is not a valid justification—it is an admission of choosing the dominated option.

Claim 3.52 (Universal). For ALL object-oriented systems where inheritance hierarchies exist and are accessible—including legacy codebases, dynamic languages, and functional languages with typeclasses—nominal typing is strictly optimal. Shape-based typing is a **capability sacrifice**, not an alternative with tradeoffs.

3.11.8 Discipline Optimality vs Migration Optimality

A critical distinction: **discipline optimality** (which typing paradigm has more capabilities) is independent of **migration optimality** (whether migrating an existing codebase is beneficial).

Definition 3.53 (Pareto Dominance). Discipline A Pareto dominates discipline B if: 1. A provides all capabilities of B 2. A provides at least one capability B lacks 3. The declaration cost of A is at most the declaration cost of B

Theorem 3.54 (Nominal Pareto Dominates Shape). Nominal typing Pareto dominates shape-based typing.

Proof. (Machine-checked in `discipline_migration.lean`) 1. Shape capabilities = {attributeCheck} 2. Nominal capabilities = {provenance, identity, enumeration, conflictResolution, attributeCheck} 3. Shape $\subset$ Nominal (strict subset) 4. Declaration cost: both require one class definition per interface 5. Therefore nominal Pareto dominates shape. ■

Theorem 3.55 (Dominance Does Not Imply Migration). Pareto dominance of discipline A over B does NOT imply that migrating from B to A is beneficial for all codebases.

Proof. (Machine-checked in `discipline_migration.lean`)

1. **Dominance is codebase-independent.** $D(A, B)$ ("A dominates B") is a relation on typing disciplines. It depends only on capability sets: Capabilities(A) $\supset$ Capabilities(B). This is a property of the disciplines themselves, not of any codebase.
2. **Migration cost is codebase-dependent.** Let $C(ctx)$ be the cost of migrating codebase ctx from B to A. Migration requires modifying: type annotations using B-specific constructs, call sites relying on B-specific semantics, and external API boundaries (which may be immutable). Each of these quantities is unbounded: there exist codebases with arbitrarily many annotations, call sites, and external dependencies.
3. **Benefit is bounded.** The benefit of migration is the capability gap: $|$Capabilities(A) $\setminus$ Capabilities(B)$|$. For nominal vs structural, this is 4 (provenance, identity, enumeration, conflict resolution). This is a constant, independent of codebase size.
4. **Unbounded cost vs bounded benefit.** For any fixed benefit B, there exists a codebase ctx such that $C(ctx) > B$. This follows from (2) and (3): cost grows without bound, benefit does not.
5. **Existence of both cases.** For small ctx: $C(ctx) < B$ (migration beneficial). For large ctx: $C(ctx) > B$ (migration not beneficial).

Therefore dominance does not determine migration benefit. ■

Corollary 3.55a (Category Error). Conflating "discipline A is better" with "migrate to A" is a category error: the former is a property of disciplines (universal), the latter is a property of (discipline, codebase) pairs (context-dependent).

Corollary 3.56 (Discipline vs Migration Independence). The question "which discipline is better?" (answered by Theorem 3.54) is independent of "should I migrate?" (answered by cost-benefit analysis).

This distinguishes "nominal provides more capabilities" from "rewrite everything in nominal." The theorems are:

- **Discipline comparison**: Universal, always true (Theorem 3.54)
- **Migration decision**: Context-dependent, requires cost-benefit analysis (Theorem 3.55)

3.11.9 Context Formalization: Greenfield and Retrofit (Historical)

Note. The following definitions were used in earlier versions of this paper to distinguish contexts where nominal typing was "available" from those where it was not. Theorem 2.10j (Adapters) eliminates this distinction: adapters make nominal typing available in all retrofit contexts. We retain these definitions for completeness and because the Lean formalization verifies them.

Definition 3.57 (Greenfield Context). A development context is *greenfield* if: 1. All modules are internal (architect can modify type hierarchies) 2. No constraints require structural typing (e.g., JSON API compatibility)

Definition 3.58 (Retrofit Context). A development context is *retrofit* if: 1. At least one module is external (cannot modify type hierarchies), OR 2. At least one constraint requires structural typing

Theorem 3.59 (Context Classification Exclusivity). Greenfield and retrofit contexts are mutually exclusive.

Proof. (Machine-checked in `context_formalization.lean`) If a context is greenfield, all modules are internal and no constraints require structural typing. If any module is external or any constraint requires structural typing, the context is retrofit. These conditions are mutually exclusive by construction. ■

Corollary 3.59a (Retrofit Does Not Imply $\{S\}$-Only). A retrofit context does not require $\{S\}$-only typing. Adapters (Theorem 2.10j) make $\{B, S\}$ typing available in all retrofit contexts where $B \neq \emptyset$.

Definition 3.60 (Provenance-Requiring Query). A system query *requires provenance* if it needs to distinguish between structurally equivalent types. Examples: - "Which type provided this value?" (provenance) - "Is this the same type?" (identity) - "What are all subtypes?" (enumeration) - "Which type wins in MRO?" (conflict resolution)

Theorem 3.61 (Provenance Detection). Whether a system requires provenance is decidable from its query set.

Proof. (Machine-checked in `context_formalization.lean`) Each query type is classified as requiring provenance or not. A system requires provenance iff any of its queries requires provenance. This is a finite check over a finite query set. ■

Theorem 3.62 (Decision Procedure Soundness). The discipline selection procedure is sound: 1. If $B \neq \emptyset \to$ select $\{B, S\}$ (dominance, universal) 2. If $B = \emptyset \to$ select $\{S\}$ (no alternative exists)

Proof. (Machine-checked in `context_formalization.lean`) Case 1: When $B \neq \emptyset$, $\{B, S\}$ typing strictly dominates $\{S\}$-only typing (Theorem 3.5). Adapters eliminate the retrofit exception (Theorem 2.10j). Therefore $\{B, S\}$ is always correct. Case 2: When $B = \emptyset$ (e.g., Go interfaces, JSON objects), $\{B, S\}$ typing is undefined—there is no inheritance to track. $\{S\}$ is the only coherent discipline. ■

Remark: In type system terminology, $\{B, S\}$ corresponds to nominal typing and $\{S\}$ to structural/duck typing.

Remark (Obsolescence of Greenfield/Retrofit Distinction). Earlier versions of this paper distinguished "greenfield" (use nominal) from "retrofit" (use shape). Theorem 2.10j eliminates this distinction: adapters make $\{B, S\}$ typing available in all retrofit contexts. The only remaining distinction is whether B exists at all.

3.6 Axis-Parametric Type Theory

The (B, S) model generalizes to a parametric framework where axis sets are *derived* from domain requirements rather than enumerated. This transforms typing discipline selection from preference to computation.

Definition 3.80 (Axis). An axis A is a recursive lattice structure:

- Carrier type with partial order
- Recursive self-reference: $A \to \text{List } A$ (e.g., bases have bases)
- Provides capabilities not derivable from other axes

Definition 3.81 (Axis Independence). Axis A is independent of axis set $\mathcal{A}$ iff $\exists$ query q such that A can answer q but no projection from $\mathcal{A}$ can answer q.

Theorem 3.82 (Axis Capability Monotonicity). For any axis set $\mathcal{A}$ and independent axis X:

$$\text{capabilities}(\mathcal{A} \cup \{X\}) \supsetneq \text{capabilities}(\mathcal{A})$$

Proof. By independence, $\exists q$ that $\mathcal{A}$ cannot answer but X can. Adding X enables q while preserving all existing capabilities. ■

Remark 3.82a (Relative vs. Ultimate Dominance). Nominal typing strictly dominates any system with *fewer* axes—but not those with more. (B, S, H) dominates (B, S) which dominates (S). Dominance is relative to axis count. The ultimately correct choice is axis-parametric: supporting N axes, where the minimal N is derived from domain requirements.

Remark 3.82b (Support vs. Usage—The Core Insight). The question is not *how many axes to use*, but *how many to support*. This parallels function parameters: a function that accepts N parameters but is called with 1 does the same work as a function that can only accept 1. The capability exists without cost until invoked.

Formally: Let $\text{work}(A, k)$ denote the work performed when using k axes from a system supporting axis set A. Then:

$$\text{work}(\{A_1, \ldots, A_N\}, 1) = \text{work}(\{A_1\}, 1)$$

Using 1 axis in an N-axis system costs exactly the same as using 1 axis in a 1-axis system. But if requirements later demand 2 axes, the N-axis system accommodates; the 1-axis system cannot. This is the *zero-cost extensibility* principle: parametric systems pay nothing for unused capability, but gain the option to extend.

Corollary 3.82c (Fixed-Axis is Dominated). Any fixed-axis system is dominated by a parametric system supporting at least those axes. The fixed system has identical capability when those axes suffice, but zero capability when additional axes are required. The parametric system is never worse and sometimes strictly better.

Theorem 3.83 (Derivability Collapse). If axis N is derivable from axis B (i.e., $\exists f : B \to N$ preserving structure), then N is not independent and any minimal axis set excludes N.

Proof. Any query answerable by N is answerable by $f(B)$. Therefore N provides no capability beyond B. ∎

Corollary 3.84 (Type Identity Collapse). Type identity is derivable from Bases B: the identity of type T is head(MRO(T)), which is T itself—the leaf of the bases graph. Therefore type identity is not an independent axis. Note: the syntactic name (`__name__`) is metadata external to the semantic model; it is not derivable from (B, S) and is not part of the minimal typing model. See Definition 2.7.

Theorem 3.85 (Completeness Uniqueness). For any domain D with requirements Q, if $\mathcal{A}_1$ and $\mathcal{A}_2$ are both minimal complete for D, then $\mathcal{A}_1 \cong \mathcal{A}_2$ (isomorphic as axis sets).

Proof. Suppose $\mathcal{A}_1 \neq \mathcal{A}_2$. WLOG $\exists A \in \mathcal{A}_1, A \notin \mathcal{A}_2$. By minimality of $\mathcal{A}_1$, $\exists q$ requiring A. By completeness of $\mathcal{A}_2$, some axis in $\mathcal{A}_2$ answers q. That axis must be isomorphic to A (answers same queries). Contradiction. ∎

Theorem 3.86 (Axis Derivation Algorithm). The minimal complete axis set for domain D is computable:

```
derive(Q):
  A := {}
  for q in Q:
    if not answerable(A, q):
      A := A + {minimal_axis_for(q)}
  return collapse(A)  -- remove derivable axes
```

The result is unique, minimal, and complete.

Empirical Validation (Inductive Evidence). The framework is validated by observed capability increments:

Transition	Axis Added	New Capabilities	Observed
$\emptyset \to (S)$	S	interface checking	Structural typing
$(S) \to (B, S)$	B	provenance, identity, enumeration, conflict	Nominal typing
$(B, S) \to (B, S, H)$	H	scope resolution, hierarchical provenance	OpenHCS

Each transition adds exactly the capabilities predicted by Theorem 3.82.

Remark 3.87 (Duck Typing is Not a Level). Duck typing is *incoherent* use of S (Theorem 2.10d), not a valid point in the axis lattice. The progression $\emptyset < (S) < (B, S) < (B, S, H)$ consists entirely of *coherent* typing disciplines. Duck typing is a failure mode, not an alternative.

Theorem 3.88 (Non-Recursive Collapse). Any non-recursive structure (e.g., flat enums, constants) is derivable from S as an attribute value. Only recursive structures can be independent axes.

Proof. Non-recursive values are atomic and can be stored as attribute values in the namespace. Therefore they are captured by S and provide no independent capability. ∎

Definition 3.89 (Preference Position). A *preference position* for domain D is the claim that multiple distinct typing disciplines are valid choices:

$$\text{Preference}(D) \triangleq \exists A_1, A_2 : \text{AxisSet. minimal}(A_1, D) \wedge \text{minimal}(A_2, D) \wedge A_1 \not\cong A_2$$

Theorem 3.90 (Preference Incoherence). For any domain D with specified requirements, the preference position is false:

$$\forall D.\ \neg\text{Preference}(D)$$

Proof. By Theorem 3.85 (Completeness Uniqueness), if A_1 and A_2 are both minimal complete for D, then $A_1 \cong A_2$. Therefore the conjunction $\mathrm{minimal}(A_1, D) \wedge \mathrm{minimal}(A_2, D) \wedge A_1 \ncong A_2$ is unsatisfiable. ■

Corollary 3.91 (Hedging Incoherence). Let R be a reader who accepts Theorems 3.80-3.88. The position "these theorems are correct but typing discipline selection remains a matter of preference or tradeoffs" is logically inconsistent.

Proof. "Preference" presupposes $\mathrm{Preference}(D)$ for some domain D. By Theorem 3.90, $\neg\mathrm{Preference}(D)$ for all D. Accepting the theorems while asserting preference instantiates $P \wedge \neg P$. ■

Remark 3.92 (Why This Theorem Exists). Empirically, readers who accept formal results often hedge on their implications ("interesting contribution, but..."). This hedging is not mere caution—it is a logical error. Corollary 3.91 makes this error explicit: one cannot coherently accept uniqueness while maintaining that alternatives exist. The debate over typing disciplines is not "contributed to" by this work; it is *resolved*.

3.7 Summary: Completeness Analysis

Potential Concern	Formal Analysis
"Model is incomplete"	Theorem 3.32 (Model Completeness)
"Duck typing has tradeoffs"	Theorems 3.34-3.36 (Capability Comparison)
"Axioms are assumptive"	Lemma 3.37 (Axiom is Definitional)
"Clever extension could fix it"	Theorem 3.39 (Extension Impossibility)
"Pack B into S to avoid adding axis"	Theorem 3.22h (Packing Dichotomy)
"What about generics?"	Theorems 3.43-3.48, Table 2.2 (Parameterized N)
"Erasure changes things"	Theorems 3.46-3.47 (Compile-Time Type Checking)
"Only works for some languages"	Theorem 3.47 (8 languages), Remark 3.49 (exotic features)
"What about intersection/union types?"	Remark 3.49 (still three axes)
"What about row polymorphism?"	Remark 3.49 (pure S, loses capabilities)
"What about higher-kinded types?"	Remark 3.49 (parameterized N)
"Only applies to greenfield"	Theorem 2.10j (Adapters eliminate retrofit exception)
"Legacy codebases are different"	Corollary 3.51 (sacrifice, not alternative)
"Claims are too broad"	Theorem 2.6 (axis requirements are domain-derived)
"You can't say rewrite everything"	Theorem 3.55 (Dominance $\neq$ Migration)
"Greenfield is undefined"	Definitions 3.57-3.58, Theorem 3.59
"Provenance requirement is circular"	Theorem 3.61 (Provenance Detection)
"Duck typing is coherent"	Theorem 2.10d (Incoherence)
"Protocol is valid for retrofit"	Theorem 2.10j (Dominated by Adapters)
"Avoiding adapters is a benefit"	Corollary 2.10k (Negative Value)
"Protocol has equivalent metaprogramming"	Theorem 2.10p (Hooks Require Declarations)
"You can enumerate Protocol implementers"	Theorem 2.10q (Enumeration Requires Registration)
"Interesting but not paradigm-shifting"	Corollary 3.91 (Hedging Incoherence)
"There are still tradeoffs"	Theorem 3.90 (Preference Incoherence)

Completeness. Appendix A provides detailed analysis of each potential concern, demonstrating why none affect our conclusions. The analysis covers model completeness, capability characterization, scope boundaries, and the distinction between discipline dominance and migration recommendation.

---

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 ($\{B, S\}$ Typing Complexity). E($\{B, S\}$) = O(1).

Proof. Under $\{B, S\}$ 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. ∎

Remark: In type system terminology, $\{B, S\}$ typing is called nominal typing.

Theorem 4.2 ($\{S\}$-Only Declared Complexity). E($\{S\}$-declared) = O(k) where k = number of classes.

Proof. Under $\{S\}$-only typing with declared interfaces, 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. ∎

Remark: In type system terminology, this is called structural typing.

Theorem 4.3 ($\{S\}$-Only Incoherent Complexity). E($\{S\}$-incoherent) = $\Omega(n)$ where n = number of call sites.

Proof. Under $\{S\}$-only incoherent 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 $\Omega(n)$. ∎

Remark: This incoherent pattern is traditionally called "duck typing."

Corollary 4.4 (Strict Dominance). $\{B, S\}$ typing strictly dominates $\{S\}$-only incoherent: E($\{B, S\}$) = O(1) < $\Omega(n)$ = E($\{S\}$-incoherent) for all n > 1.

Remark: In type system terminology, this shows nominal typing dominates duck typing.

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 ($\{S\}$-Only Incoherent Scattering). For $\{S\}$-only incoherent typing, |I($\{S\}$-incoherent, c)| = O(n) where n = call sites using constraint c.

Remark: This describes the scattering problem in "duck typing."

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

Theorem 4.6 ($\{B, S\}$ Typing Centralizes). For $\{B, S\}$ typing, |I($\{B, S\}$, 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. ∎

Remark: In type system terminology, $\{B, S\}$ typing is called nominal typing.

Corollary 4.7 (Maintenance Entropy). $\{S\}$-only incoherent typing maximizes maintenance entropy; $\{B, S\}$ typing minimizes it.

Remark: Traditional terms: duck typing vs nominal typing.

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 $\{B, S\}$ typing but Ω(n) call-site inspection under $\{S\}$-only incoherent typing. Case Study 5 illustrates this: migrating from incoherent to $\{B, S\}$ typing replaced scattered `hasattr()` checks across 47 call sites with centralized ABC contract validation at a single definition point.

Remark: In type system terminology, $\{B, S\}$ is nominal typing; $\{S\}$-only incoherent is duck typing.

5 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 [35]). We address this in three ways:

First: Claim structure. This paper makes two distinct types of claims with different validation requirements. *Mathematical claims* (Theorems 3.1–3.62): "Discarding B necessarily loses these capabilities." These are proven by formal derivation in Lean (6300+ lines, 0 `sorry`). Mathematical proofs have no sample size: they are universal by construction. *Existence claims*: "Production systems requiring these capabilities exist." One example suffices for an existential claim. OpenHCS demonstrates that real systems require provenance tracking, MRO-based resolution, and type-identity dispatch, exactly the capabilities Theorem 3.19 proves impossible under $\{S\}$-only typing.

Remark: In type system terminology, $\{S\}$-only typing is called structural typing.

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 $\{B, S\}$ typing; the case studies confirm this prediction. The 13 patterns are 13 independent architectural decisions, each of which could have used $\{S\}$-only typing but provably could not. Packaging these patterns into separate repositories would not add information: it would be technicality theater. The mathematical impossibility results are the contribution; OpenHCS is the existence proof that the impossibility matters.

Remark: In type system terminology, $\{B, S\}$ is nominal typing; $\{S\}$-only is structural typing.

Third: Falsifiable predictions. The decision procedure (Theorem 3.62) makes falsifiable predictions: systems where $B \neq \emptyset$ should exhibit $\{B, S\}$ patterns; systems where $B = \emptyset$ should exhibit $\{S\}$-only patterns. Any codebase where this prediction fails would falsify our theory.

Remark: In type system terminology, $\{B, S\}$ patterns are nominal; $\{S\}$-only are structural.

The validation structure:

Level	What it provides	Status
Formal proofs	Mathematical necessity	Complete (Lean, 6300+ lines, 0 `sorry`)
OpenHCS case studies	Existence proof	13 patterns documented
Decision procedure	Falsifiability	Theorem 3.62 (machine-checked)

OpenHCS is a bioimage analysis platform for high-content screening microscopy. The system was designed from the start with explicit commitment to $\{B, S\}$ 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 $\{B, S\}$ typing is mandatory per the formal model.

Remark: In type system terminology, $\{B, S\}$ typing is called nominal typing.

$\{S\}$-only incoherent 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 $\{S\}$-only typing's structural equivalence axiom.

Remark: "$\{S\}$-only incoherent typing" is traditionally called "duck typing."

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 ↔ base type registries). Table 5.2 summarizes how each pattern fails under $\{S\}$-only incoherent typing and what $\{B, S\}$ mechanism enables it.

Remark: Traditional terms: duck typing (incoherent) and nominal typing.

5.1.1 Table 5.1: Case Studies as Theorem Validation

Study	Pattern	Validates Theorem	Validation Type
1	Type discrimination	Theorem 3.4 (Nominal Pareto-Dominance)	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 4.1 (O(1) Complexity)	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 (Nominal Pareto-Dominance)	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

5.1.2 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
7	Type transformation	Cannot track lineage	5-stage `type()` chain
8	Dual-axis resolution	No scope × MRO product	Registry + MRO traversal

Study	Pattern	Duck Failure Mode	Nominal Mechanism
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

Theorem 5.1 (Structural Identity $\neq$ Semantic Identity). Two types A and B with identical structure $S(A) = S(B)$ may have distinct semantics determined by their position in an inheritance hierarchy. Duck typing's axiom of structural equivalence ($S(A) = S(B) \Rightarrow A \equiv B$) destroys this semantic distinction.

Proof. By construction from production code.

The Diamond Inheritance Pattern:

```
                    WellFilterConfig
                   (well_filter, well_filter_mode)
                  /                              \
                 /                                \
    PathPlanningConfig                     StepWellFilterConfig
    (output_dir_suffix,                          (pass)
     global_output_folder,                [NO NEW FIELDS - STRUCTURALLY
     sub_dir = "images")                   IDENTICAL TO WellFilterConfig]
                 \                                /
                  \                              /
                   StepMaterializationConfig
                   (sub_dir = "checkpoints", enabled)
```

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

@dataclass(frozen=True)
class PathPlanningConfig(WellFilterConfig):
    """Pipeline-level path configuration."""
    output_dir_suffix: str = "_openhcs"
    sub_dir: str = "images"  # Pipeline default

@dataclass(frozen=True)
class StepWellFilterConfig(WellFilterConfig):
    """Step-level scope marker."""
    pass  # ZERO new fields. Structurally identical to WellFilterConfig.

@dataclass(frozen=True)
class StepMaterializationConfig(StepWellFilterConfig, PathPlanningConfig):
    """Step-level materialization."""
```

```
    sub_dir: str = "checkpoints"  # Step default OVERRIDES pipeline default
    enabled: bool = False
```

Critical observation: `StepWellFilterConfig` adds **zero fields**. It is byte-for-byte structurally identical to `WellFilterConfig`. Yet it serves a critical semantic role: it marks the **scope boundary** between pipeline-level and step-level configuration.

The MRO encodes scope semantics:

```
StepMaterializationConfig.__mro__ = (
    StepMaterializationConfig,  # Step scope
    StepWellFilterConfig,       # Step scope marker (NO FIELDS!)
    PathPlanningConfig,         # Pipeline scope
    WellFilterConfig,           # Pipeline scope
    object
)
```

When resolving `sub_dir`: 1. `StepMaterializationConfig.sub_dir = "checkpoints"` → **step-level value** 2. `PathPlanningConfig.sub_dir = "images"` → pipeline-level value (shadowed)

The system answers "which scope provided this value?" by walking the MRO. The *position* of `StepWellFilterConfig` (before `PathPlanningConfig`) encodes the scope boundary.

What duck typing sees:

Object	`well_filter`	`well_filter_mode`	`sub_dir`
WellFilterConfig()	None	INCLUDE	—
StepWellFilterConfig()	None	INCLUDE	—

Duck typing's verdict: **identical**. Same attributes, same values.

What the system needs to know:

1. "Is this config pipeline-level or step-level?" → Determines resolution priority
2. "Which type in the MRO provided `sub_dir`?" → Provenance for debugging
3. "Can I use `isinstance(config, StepWellFilterConfig)`?" → Scope discrimination

$\{S\}$-only typing cannot answer ANY of these questions. The information is **not in the structure**: it is in the **type identity** and **MRO position**.

$\{B, S\}$ typing answers all three in O(1):

Remark: In type system terminology, $\{B, S\}$ is nominal typing; $\{S\}$-only is duck/structural typing.

```
isinstance(config, StepWellFilterConfig)  # Scope check: O(1)
type(config).__mro__                        # Full provenance chain: O(1)
type(config).__mro__.index(StepWellFilterConfig)  # MRO position: O(k)
```

Corollary 5.2 (Scope Encoding Requires $\{B, S\}$ Typing). Any system that encodes scope semantics in inheritance hierarchies (where structurally-identical types at different MRO positions have different meanings) **requires** $\{B, S\}$ typing. $\{S\}$-only typing makes such architectures impossible (not difficult, **impossible**).

Remark: In type system terminology, $\{B, S\}$ typing is called nominal typing; $\{S\}$-only is duck/structural typing.

Proof. $\{S\}$-only typing defines equivalence as $S(A) = S(B) \Rightarrow A \equiv B$. If A and B are structurally identical but semantically distinct (different scopes), $\{S\}$-only typing **by definition** cannot distinguish them. This is not a limitation of $\{S\}$-only implementations; it is the **definition** of $\{S\}$-only typing. ■

This is not an edge case. The OpenHCS configuration system has 15 `@global_pipeline_config` decorated dataclasses forming multiple diamond inheritance patterns. The entire architecture depends on

MRO position distinguishing types with identical structure. Under $\{S\}$-only typing, this system **cannot exist**.

Remark: "Duck typing" is the traditional term for $\{S\}$-only incoherent typing.

Pattern (Table 5.1, Row 1): Type discrimination via MRO position. This case study demonstrates: - Theorem 4.1: O(1) type identity via `isinstance()` - Theorem 4.3: O(1) vs $\Omega(n)$ complexity gap - The fundamental failure of structural equivalence to capture semantic distinctions

5.2.1 Sentinel Attribute Objection

Objection: "Just add a sentinel attribute (e.g., `_scope: str = 'step'`) to distinguish types structurally."

Theorem 5.2a (Sentinel Attribute Insufficiency). Let $\sigma : T \to V$ be a sentinel attribute (a structural field intended to distinguish types). Then σ cannot recover any B-dependent capability.

Proof. 1. **Sentinel is structural.** By definition, σ is an attribute with a value. Therefore $\sigma \in S(T)$ (the structure axis). 2. **B-dependent capabilities require B.** By Theorem 3.19, provenance, identity, enumeration, and conflict resolution all require the Bases axis B. 3. **S does not contain B.** By the axis independence property (Definition 2.5), the axes (B, S) are independent: S carries no information about B. 4. **Therefore σ cannot provide B-dependent capabilities.** Since $\sigma \in S$ and B-dependent capabilities require information not in S, no sentinel attribute can recover them. ■

Corollary 5.2b (Specific Sentinel Failures).

Capability	Why sentinel fails
Enumeration	Requires iterating over types with $\sigma = v$. No type registry exists in structural typing (Theorem 2.10q). Cannot compute `[T for T in ? if T._scope == 'step']`. There is no source for `?`.
Enforcement	σ is a runtime value, not a type constraint. Subtypes can set σ incorrectly without type error. No enforcement mechanism exists.
Conflict resolution	When multiple mixins define σ, which wins? This requires MRO, which requires B. Sentinel $\sigma \in S$ has no MRO.
Provenance	"Which type provided σ?" requires MRO traversal. σ cannot answer queries about its own origin.

Corollary 5.2c (Sentinel Simulates, Cannot Recover). Sentinel attributes can *simulate* type identity (by convention) but cannot *recover* the capabilities that identity provides. The simulation is unenforced (violable without type error), unenumerable (no registry), and unordered (no MRO for conflicts). This is precisely the capability gap of Theorem 3.19, repackaged. ■

5.2.1 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 is 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.

5.3 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.

5.4 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. $\{B, S\}$ typing couples the check to type identity, which is refactoring-safe.

Remark: In type system terminology, $\{B, S\}$ is nominal typing.

Pattern (Table 5.1, Row 4): Polymorphic dispatch with interface guarantees. Demonstrates how $\{B, S\}$ ABC contracts provide fail-loud validation that $\{S\}$-only incoherent typing's fail-silent probing cannot match.

Remark: Traditional terms: nominal typing vs duck typing.

5.5 Case Study 5: Migration from $\{S\}$-Only Incoherent to $\{B, S\}$ Typing (PR #44)

Remark: Traditional terms: duck typing to nominal typing.

PR #44 [36] ("UI Anti-Duck-Typing Refactor") migrated OpenHCS's UI layer from $\{S\}$-only incoherent typing to $\{B, S\}$ ABC contracts. The architectural changes:

Before ($\{S\}$-only incoherent): - ParameterFormManager: 47 `hasattr()` dispatch points scattered across methods - CrossWindowPreviewMixin: attribute-based widget probing throughout - Dispatch tables: string attribute names mapped to handlers

After ($\{B, S\}$ typing): - ParameterFormManager: single `AbstractFormWidget` ABC with explicit contracts - CrossWindowPreviewMixin: explicit widget protocols - Dispatch tables: eliminated, replaced by `isinstance()` + method calls

Architectural transformation:

```
# BEFORE: {S}-only incoherent dispatch (scattered across 47 call sites)
if hasattr(widget, 'isChecked'):
    return widget.isChecked()
elif hasattr(widget, 'currentText'):
    return widget.currentText()
# ... 45 more cases

# AFTER: {B,S} 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 $\{S\}$-only incoherent typing to fail-loud $\{B, S\}$ contracts. Demonstrates the complexity reduction predicted by Theorem 4.3: scattered `hasattr()` checks (n=47) were replaced with O(1) centralized ABC

Remark: Traditional terms: duck typing to nominal typing. validation.

5.6 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 -> auto-registered
class ImageXpressHandler(MicroscopeHandler, metaclass=MicroscopeHandlerMeta):
    _microscope_type = 'imagexpress'
```

This pattern is impossible with $\{S\}$-only incoherent typing because: (1) type identity is required as dict values. $\{S\}$-only incoherent 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 $\{S\}$-only incoherent typing's instance-level probing, and (3) inheritance-based key derivation (extracting "imagexpress" from "ImageXpressHandler") requires class name access.

Remark: Traditional term: duck typing.

The metaclass ensures exactly one handler per microscope type. Attempting to define a second `ImageXpressHandler` raises an exception at import time. $\{S\}$-only incoherent typing's runtime checks cannot provide this guarantee. Duplicates would silently overwrite.

Remark: Traditional term: duck typing.

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

Remark: Traditional term: nominal typing.

5.7 Case Study 7: Five-Stage Type Transformation

The decorator chain demonstrates $\{B, S\}$ typing's power for systematic type manipulation. Starting from `@auto_create_decorator`, one decorator invocation spawns a cascade that generates lazy companion types, injects fields into parent configs, and maintains bidirectional registries.

Remark: Traditional term: nominal typing.

Stage 1: `@auto_create_decorator` on `GlobalPipelineConfig`

```
@auto_create_decorator
@dataclass(frozen=True)
```

```
class GlobalPipelineConfig:
    num_workers: int = 1
```

The decorator: 1. Validates naming convention (must start with "Global") 2. Marks class: `global_config_class._is_global_config = True` 3. Calls `create_global_default_decorator(GlobalPipelineConfig)` → returns `global_pipeline_config` 4. Exports to module: `setattr(module, 'global_pipeline_config', decorator)`

Stage 2: `@global_pipeline_config` applied to nested configs

```
@global_pipeline_config(inherit_as_none=True)
@dataclass(frozen=True)
class PathPlanningConfig(WellFilterConfig):
    output_dir_suffix: str = ""
```

The generated decorator: 1. If `inherit_as_none=True`: rebuilds class with `None` defaults for inherited fields via `rebuild_with_none_defaults()` 2. Generates lazy class: `LazyDataclassFactory.make_lazy_simple(PathPlanningConfig, "LazyPathPlanningConfig")` 3. Exports lazy class to module: `setattr(config_module, "LazyPathPlanningConfig", lazy_class)` 4. Registers for pending field injection into `GlobalPipelineConfig` 5. Binds lazy resolution to concrete class via `bind_lazy_resolution_to_class()`

Stage 3: Lazy class generation via `make_lazy_simple`

Inside `LazyDataclassFactory.make_lazy_simple()`: 1. Introspects base class fields via `_introspect_dataclass_fields()` 2. Creates new class: `make_dataclass("LazyPathPlanningConfig", fields, bases=(PathPlanningConfig, LazyDataclass))` 3. Registers bidirectional type mapping: `register_lazy_type_mapping(lazy_class, base_class)`

Stage 4: Field injection via `_inject_all_pending_fields`

At module load completion: 1. Collects all pending configs registered by `@global_pipeline_config` 2. Rebuilds `GlobalPipelineConfig` with new fields: `path_planning: LazyPathPlanningConfig = field(default_factory=LazyPathPlanningConfig)` 3. Preserves `_is_global_config = True` marker on rebuilt class

Stage 5: Resolution via MRO + context stack

At runtime, dual-axis resolution walks `type(config).__mro__`, normalizing each type via registry lookup. The `sourceType` in `(value, scope, sourceType)` carries provenance that duck typing cannot provide.

Nominal typing requirements throughout: - Stage 1: `_is_global_config` marker enables `isinstance(obj, GlobalConfigBase)` via metaclass - Stage 2: `inherit_as_none` marker controls lazy factory behavior - Stage 3: `type()` identity in bidirectional registries - Stage 4: `type()` identity for field injection targeting - Stage 5: MRO traversal requires `B` axis

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 $\{B, S\}$ identities.

Remark: Traditional term: nominal identities.

5.8 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 $\rightarrow$ plate $\rightarrow$ step) and MRO (child class $\rightarrow$ 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 $\{B, S\}$ typing where `PathPlanningConfig` and `LazyPathPlanningConfig` are distinct despite identical structure. $\{S\}$-only incoherent typing sees both as having the same attributes, making `mro_type` meaningless.

Remark: Traditional terms: nominal typing vs duck typing.

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

Remark: Traditional term: duck typing.

Pattern (Table 5.1, Row 8): Dual-axis resolution with scope $\times$ MRO product. Demonstrates that provenance tracking fundamentally requires $\{B, S\}$ identity (Corollary 6.3).

Remark: Traditional term: nominal identity.

5.9 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 $\{B, S\}$ marker that $\{S\}$-only incoherent typing cannot replicate.

Remark: Traditional terms: nominal marker vs duck typing.

$\{S\}$-only incoherent 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 $\{B, S\}$: the check is "does this type have this marker?" not "does this instance have this attribute?"

Remark: Traditional terms: duck typing vs nominal.

The virtual inheritance enables interface segregation: `GlobalPipelineConfig` advertises conformance to `GlobalConfigBase` without inheriting implementation. This is impossible with $\{S\}$-only incoherent typing's attribute probing. There's no way to express "this class satisfies this interface" as a runtime-checkable property.

Remark: Traditional term: duck typing.

Pattern (Table 5.1, Row 9): Custom isinstance via class-level markers. Demonstrates that Python's metaobject protocol is fundamentally $\{B, S\}$.

Remark: Traditional term: nominal.

5.10 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 {B,S} 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) -> True
```

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

Remark: Traditional terms: nominal identity, pure nominal typing, structural typing vs nominal typing.

$\{S\}$-only incoherent typing has no equivalent concept. There's no way to express "this class explicitly implements this contract" without actual attributes to probe. The $\{B, S\}$ marker enables explicit interface declarations in a dynamically-typed language.

Remark: Traditional terms: duck typing vs nominal marker.

Pattern (Table 5.1, Row 10): Runtime-generated interfaces with empty structure. Demonstrates that $\{B, S\}$ identity can exist independent of structural content.

Remark: Traditional term: nominal identity.

5.11 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') -> fragile, module probing
# 'framework' in config_names -> string-based, no type safety
```

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

Remark: Traditional term: nominal identity.

$\{S\}$-only incoherent 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 $\{B, S\}$ sentinel provides a refactoring-safe, type-safe marker that exists independent of names or attributes.

Remark: Traditional terms: duck typing vs nominal.

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 $\{B, S\}$ identity as a capability marker independent of structure.

Remark: Traditional term: nominal identity.

5.12 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. $\{S\}$-only incoherent typing has no concept of "the type itself" as a mutable namespace. It can only access instances. To inject methods $\{S\}$-incoherent-style would require modifying every instance's `__dict__`, which doesn't affect future instances.

Remark: Traditional term: duck typing.

The $\{B, S\}$ 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 $\{B, S\}$: the type is a first-class object with its own namespace, distinct from instance namespaces.

Remark: Traditional term: nominal.

This pattern enables plugins, mixins, and monkey-patching. All requiring types as mutable namespaces. $\{S\}$-only incoherent typing's instance-level view cannot express "modify the behavior of all objects of this kind."

Remark: Traditional term: duck typing.

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 $\{B, S\}$ identity.

Remark: Traditional term: nominal identity.

5.13 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 -> base
        self._reverse: Dict[Type, Type] = {}  # base -> 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) -> PathPlanningConfig
#        registry.get_lazy(PathPlanningConfig) -> LazyPathPlanningConfig
```

$\{S\}$-only incoherent typing would require maintaining two separate dicts with string keys (class names), introducing synchronization bugs. Renaming `PathPlanningConfig` would break the string-based lookup. The

Remark: Traditional term: duck typing. 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 (6300+ lines across eleven modules, 0 `sorry` placeholders) is organized as follows:

Module	Lines	Theorems/Lemmas	Purpose
`abstract_class_system.lean`	3082	90+	Core formalization: two-axis model, dominance, complexity
`axis_framework.lean`	1667	40+	Axis parametric theory, matroid structure, fixed-axis incompleteness
`nominal_resolution.lean`	556	21	Resolution, capability exhaustiveness, adapter amortization
`discipline_migration.lean`	142	11	Discipline vs migration optimality separation
`context_formalization.lean`	215	7	Greenfield/retrofit classification, requirement detection
`python_instantiation.lean`	247	12	Python-specific instantiation of abstract model
`typescript_instantiation.lean`	65	3	TypeScript instantiation

Module	Lines	Theorems/Lemmas	Purpose
`java_instantiation.lean`	63	3	Java instantiation
`rust_instantiation.lean`	64	3	Rust instantiation
Total	**6100+**	**190+**	

1. **Language-agnostic layer** (Section 6.12): The two-axis model (B, S), axis lattice metatheorem, and strict dominance: proving nominal typing dominates shape-based typing in **any** class system with explicit inheritance. These proofs require no Python-specific axioms.
2. **Python instantiation layer** (Sections 6.1–6.11): The dual-axis resolution algorithm, provenance preservation, and OpenHCS-specific invariants: proving that Python's `type(name, bases, namespace)` and C3 linearization correctly instantiate the abstract model.
3. **Complexity bounds layer** (Section 6.13): Formalization of O(1) vs O(k) vs $\Omega(n)$ complexity separation. Proves that nominal error localization is O(1), structural is O(k), duck is $\Omega(n)$, and the gap grows without bound.

The abstract layer establishes that our theorems apply to Java, C#, Ruby, Scala, and any language with the (B, S) structure. The Python layer demonstrates concrete realization. The complexity layer proves the asymptotic dominance is machine-checkable, not informal.

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 -> 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 -> 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 -> 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 != 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 != 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 <->
    (v = 0 /\ resolve R mro scopes ctx = none) \/
    (exists r : ResolveResult,
      resolve R mro scopes ctx = some r /\ 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 [<- 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 ->
             resolve R mro scopes ctx = r_2 ->
             r_1 = r_2 := by
  intros r_1 r_2 h_1 h_2
  rw [<- h_1, <- 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 * 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 -> structurallyEquivalent b a := by
  intro h name; exact (h name).symm

theorem structEq_trans (a b c : DuckObject) :
    structurallyEquivalent a b -> structurallyEquivalent b c ->
    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 follows from 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 -> a) : Prop :=
  forall a b, structurallyEquivalent a b -> 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 -> 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 -> 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 != 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 (B, S)	80	PASS Definitions
- Typing discipline capabilities	100	PASS Proved
- Strict dominance (Theorem 2.18)	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
MetaprogrammingGap namespace	156	PASS Compiles, no warnings
- Declaration/Query/Hook definitions	30	PASS Definitions
- Theorem 2.10p (Hooks Require Declarations)	20	PASS Proved
- Structural typing model	35	PASS Definitions
- Theorem 2.10q (Enumeration Requires Registration)	30	PASS Proved
- Capability model & dominance	35	PASS Proved
- Corollary 2.10r (No Declaration No Hook)	15	PASS Proved
CapabilityExhaustiveness namespace	42	PASS Compiles, no warnings
- List operation/capability definitions	20	PASS Definitions
- Theorem 3.43a (capability_exhaustiveness)	12	PASS Proved
- Corollary 3.43b (no_missing_capability)	10	PASS Proved
AdapterAmortization namespace	60	PASS Compiles, no warnings
- Cost model definitions	25	PASS Definitions
- Theorem 3.43d (adapter_amortization)	10	PASS Proved
- Corollary 3.43e (adapter_always_wins)	10	PASS Proved
- Theorem (adapter_cost_constant)	8	PASS Proved
- Theorem (manual_cost_grows)	10	PASS Proved
Total	**556**	**PASS All proofs verified, 0 `sorry`, 0 warnings**

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 and complexity**: We use Lean's termination checker to verify `resolve` terminates. The complexity bound O(|scopes| $\times$ |MRO|) is also mechanically verified via `resolution_complexity_bound` and related lemmas proving linearity in each dimension.

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 On the Nature of Foundational Proofs

A reader examining the Lean source code will notice that most proofs are remarkably short, often 1-3 lines. For example, the provenance impossibility theorem (Theorem 3.13) has a one-line proof: `exact h_shape A B h_same_ns`. This brevity is not an accident or a sign of triviality. It is the hallmark of *foundational* work, where the insight lies in the formalization, not the derivation.

Definitional vs. derivational proofs. Our core theorems establish *definitional* impossibilities, not algorithmic complexities. When we prove that no shape-respecting function can compute provenance (Theorem 3.13), we are not saying "all known algorithms fail" or "the problem is NP-hard." We are saying something stronger: *it is information-theoretically impossible*. The proof follows immediately from the definition of shape-respecting functions. If two types have the same shape, any shape-respecting function must treat them identically. This is not a complex derivation; it is an unfolding of definitions.

Precedent in foundational CS. This pattern appears throughout foundational computer science:

- **Turing's Halting Problem (1936):** The proof is a simple diagonal argument, perhaps 10 lines in modern notation. Yet it establishes a fundamental limit on computation that no future algorithm can overcome.
- **Brewer's CAP Theorem (2000):** The impossibility proof is straightforward: if a partition occurs, a system cannot be both consistent and available. The insight is in the *formalization* of what consistency, availability, and partition-tolerance mean, not in the proof steps.
- **Curry-Howard Correspondence (1958/1969):** The isomorphism between types and propositions is almost definitional once the right abstractions are identified. The profundity is in recognizing the correspondence, not deriving it.

Why simplicity indicates strength. A definitional impossibility is *stronger* than a computational lower bound. Proving that sorting requires $\Omega(n \log n)$ comparisons in the worst case (decision tree argument) leaves open the possibility of non-comparison-based algorithms (radix sort, counting sort). Proving that provenance is not shape-respecting *closes all loopholes*. No algorithm, no external state, no future language feature can make shape-based typing compute provenance without abandoning the definition of "shape-based."

Where the insight lies. The semantic contribution of our formalization is threefold:

1. **Precision forcing.** Formalizing "shape-based typing" in Lean requires stating exactly what it means for a function to be shape-respecting (Definition: `ShapeRespecting`). This precision eliminates ambiguity. Informal arguments can wave hands; formal proofs cannot.
2. **Completeness guarantee.** The query space partition (Theorem 3.19) proves that *every* query is either shape-respecting or Bases-dependent. The partition is mathematical (tertium non datur), deriving the capability gap from logic.
3. **Universal scope.** The proofs apply to *any* shape-based typing discipline, not just specific implementations. The impossibility holds for duck typing (Python), structural typing (TypeScript), Protocols (PEP 544), and any future system that discards the Bases axis.

What machine-checking guarantees. The Lean compiler verifies that every proof step is valid, every definition is consistent, and no axioms are added beyond Lean's foundations. The proofs rely only on Lean 4's standard axioms (`propext`, `Quot.sound`, `Classical.choice`)—no custom axioms are introduced. Zero `sorry` placeholders means zero unproven claims. The 6300+ lines establish a verified chain from axioms to theorems. Reviewers need not trust our informal explanations. They can run `lake build` and verify the proofs themselves.

Comparison to informal arguments. Prior work on typing disciplines (Cook et al. [13], Abadi & Cardelli [1]) presents compelling informal arguments but lacks machine-checked proofs. Our contribution is not new *wisdom*. The insight that nominal typing provides capabilities structural typing lacks is old. Our contribution is *formalization*: making the argument precise enough to mechanize, closing loopholes, and proving the claims hold universally within scope.

This is the tradition of metatheory established by Liskov & Wing [21] for behavioral subtyping and Reynolds [32] for parametricity. The goal is not to prove that specific programs are correct, but to establish what is *possible* within a formal framework. Simple proofs from precise definitions are the gold standard of this work.

6.12 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 $\to$ SourceType be any external provenance map. Then:

$$\text{ExternalMap cannot answer: "Which type in MRO(type(obj)) provided attribute } a\text{?"}$$

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.

6.13 Abstract Model Lean Formalization

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

```
-- The two axes of a class system
-- NOTE: "Name" (N) is NOT an independent axis: it is metadata stored in
-- the namespace (S axis) as __name__. It contributes no typing capability.
-- The minimal model is (B, S).
inductive Axis where
  | 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 := [.Namespace]  -- S-only: structural typing (duck typing is incoherent S)
def nominalAxes : AxisSet := [.Bases, .Namespace]  -- (B, S): 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
  | .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
```

Definition 6.3a (Axis Projection — Lean). A type over axis set A is a dependent tuple of axis values:

```
-- Type as projection: for each axis in A, provide its carrier value
def AxisProjection (A : List Axis) := (a : Axis) -> a in A -> AxisCarrier a

-- Concrete Typ is a 2-tuple: (namespace, bases)
structure Typ where
  ns : Finset AttrName
  bs : List Typ
```

Theorem 6.3b (Isomorphism Theorem — Lean). The concrete type `Typ` is isomorphic to the 2-axis projection:

```
-- The Isomorphism: Typ <-> AxisProjection {B, S}
noncomputable def Typ.equivProjection : Typ <~> AxisProjection canonicalAxes where
  toFun := Typ.toProjection
  invFun := Typ.fromProjection
  left_inv := Typ.projection_roundtrip      -- fromProjection . toProjection = id
  right_inv := Typ.projection_roundtrip_inv -- toProjection . fromProjection = id

-- For any axis set A, GenericTyp A IS AxisProjection A (definitionally)
theorem n_axis_types_are_projections (A : List Axis) :
    GenericTyp A = AxisProjection A := rfl
```

This formalizes Definition 2.10 (Typing Disciplines as Axis Projections): a typing discipline using axis set A has exactly the information contained in the A-projection of the full type.

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 decide
  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 shapeAxes) :=
  <axis_shape_subset_nominal, axis_nominal_exceeds_shape>
```

This formalizes Theorem 2.15: using more axes provides strictly more capabilities. The proofs are complete and compile without any `sorry` placeholders.

Theorem 6.11 (Capability Completeness — Lean). The Bases axis provides exactly four capabilities, no more:

```
-- All possible capabilities in the system
inductive Capability where
  | interfaceCheck      -- "Does x have method m?"
  | typeNaming          -- "What is the name of type T?"
```

```
  | valueAccess         -- "What is x.a?"
  | methodInvocation    -- "Call x.m()"
  | provenance          -- "Which type provided this value?"
  | identity            -- "Is x an instance of T?"
  | enumeration         -- "What are all subtypes of T?"
  | conflictResolution  -- "Which definition wins in diamond?"
deriving DecidableEq, Repr

-- Capabilities that require the Bases axis
def basesRequiredCapabilities : List Capability :=
  [.provenance, .identity, .enumeration, .conflictResolution]

-- Capabilities that do NOT require Bases (only need N or S)
def nonBasesCapabilities : List Capability :=
  [.interfaceCheck, .typeNaming, .valueAccess, .methodInvocation]

-- THEOREM: Bases capabilities are exactly {provenance, identity, enumeration, conflictResolution}
theorem bases_capabilities_complete :
    forall c : Capability,
      (c in basesRequiredCapabilities <=>
       c = .provenance \/ c = .identity \/ c = .enumeration \/ c = .conflictResolution) := by
  intro c
  constructor
  * intro h
    simp [basesRequiredCapabilities] at h
    exact h
  * intro h
    simp [basesRequiredCapabilities]
    exact h

-- THEOREM: Non-Bases capabilities are exactly {interfaceCheck, typeNaming, valueAccess, methodInvocatio
theorem non_bases_capabilities_complete :
    forall c : Capability,
      (c in nonBasesCapabilities <=>
       c = .interfaceCheck \/ c = .typeNaming \/ c = .valueAccess \/ c = .methodInvocation) := by
  intro c
  constructor
  * intro h
    simp [nonBasesCapabilities] at h
    exact h
  * intro h
    simp [nonBasesCapabilities]
    exact h

-- THEOREM: Every capability is in exactly one category (partition)
theorem capability_partition :
    forall c : Capability,
      (c in basesRequiredCapabilities \/ c in nonBasesCapabilities) /\
      ~(c in basesRequiredCapabilities /\ c in nonBasesCapabilities) := by
  intro c
  cases c < simp [basesRequiredCapabilities, nonBasesCapabilities]

-- THEOREM: |basesRequiredCapabilities| = 4 (exactly four capabilities)
theorem bases_capabilities_count :
```

```
basesRequiredCapabilities.length = 4 := by rfl
```

This formalizes Theorem 2.17 (Capability Completeness): the capability set $\mathcal{C}_B$ is **exactly** four elements, proven by exhaustive enumeration with machine-checked partition. The `capability_partition` theorem proves that every capability falls into exactly one category (Bases-required or not) with no overlap and no gaps.

Scope as observational quotient. We model "scope" as a set of allowed observers $\mathrm{Obs} \subseteq (W \to O)$ and define observational equivalence $x \approx y \;:\Longleftrightarrow\; \forall f \in \mathrm{Obs}, f(x) = f(y)$. The induced quotient $W/{\approx}$ is the canonical object for that scope, and every in-scope observer factors through it (see `observer_factors` in `abstract_class_system.lean`). Once the observer set is fixed, no argument can appeal to information outside that quotient; adding a new observable is literally expanding Obs.

Protocol runtime observer (shape-only). We also formalize the restricted Protocol/isinstance observer that checks only for required members. The predicate `protoCheck` ignores protocol identity and is proved shape-respecting (`protoCheck_in_shapeQuerySet` in `abstract_class_system.lean`), so two protocols with identical member sets are indistinguishable to that observer. Distinguishing them requires adding an observable discriminator (brand/tag/nominality), i.e., moving to another axis.

All Python object-model observables factor through axes. In the Python instantiation we prove that core runtime discriminators are functions of (B, S): metaclass selection depends only on `bases` (`metaclass_depends_on_bases`); attribute presence and dispatch depend only on the namespace (`getattr_depends_on_ns`); together they yield `observer_factors_through_axes` in `python_instantiation.lean`.

7 Related Work

7.1 Type Theory Foundations

Malayeri & Aldrich [22, 24]. 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: when $B \neq \emptyset$ (explicit inheritance hierarchies), nominal typing is not just "appropriate" but *necessary* for capabilities like provenance tracking and MRO-based resolution. Adapters eliminate the retrofit exception (Theorem 2.10j).

Abdelgawad & Cartwright [2]. 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. [13]'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 [3]. 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. $\{S\}$-only typing loses this information after attribute access.

Remark: "Duck typing" in this context refers to $\{S\}$-only typing.

7.2 Practical Hybrid Systems

Gil & Maman [17]. 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 demonstrates the capabilities that $\{B, S\}$ typing enables: MRO-based resolution, bidirectional type registries, provenance tracking. These are impossible under $\{S\}$-only typing regardless of whether the system is new or legacy. The capability gap is information-theoretic (Theorem 3.19).

Remark: In type system terminology, $\{B, S\}$ and $\{S\}$ correspond to nominal and structural typing.

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 $\{B, S\}$ typing**. The TypeScript issue tracker has open requests for native $\{B, S\}$-style types.

Our contribution: OpenHCS avoids this problem by using $\{B, S\}$ 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.

Remark: In type system terminology, $\{B, S\}$ typing corresponds to nominal typing.

7.3 Metaprogramming Complexity

Veldhuizen [40]. "Tradeoffs in Metaprogramming" proves that sufficiently expressive metaprogramming can yield **unbounded savings** in code length. Blum [8] 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 [15]. 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 [21]. 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 Error Detection Timing

Boehm [9]. The foundational work on software engineering economics established empirical data on error cost escalation: errors detected at requirements or design time cost 1x to fix, while errors found during testing cost 10x, and post-release errors cost 50x–100x more [28]. This provides empirical grounding for why compile-time type checking matters.

Our contribution: Nominal typing shifts certain class of errors (type mismatches, spurious subtyping) from runtime to definition time. When a structural system permits an unrelated type with matching shape, the error surfaces as a logic bug during testing or production—a 10x–100x cost multiplier. Nominal typing's requirement for explicit inheritance declarations forces these mismatches to be resolved during development, paying the 1x cost. The $\Omega(n)$ vs $O(1)$ complexity separation (Theorem 4.3) formalizes this: structural systems defer error detection to n call-site inspections; nominal systems detect at the single declaration site.

7.6 Positioning This Work

7.5.1 Literature Search Methodology *Databases searched:* ACM Digital Library, IEEE Xplore, arXiv (cs.PL, cs.SE), Google Scholar, DBLP

Search terms: "nominal structural typing dominance", "typing discipline comparison formal", "structural typing impossibility", "nominal typing proof Lean Coq", "type system verification", "duck typing formalization"

Date range: 1988–2024 (Cardelli's foundational work to present)

Inclusion criteria: Peer-reviewed publications or major arXiv preprints with $\geq$10 citations; addresses nominal vs structural typing comparison with formal or semi-formal claims

Exclusion criteria: Tutorials/surveys without new theorems; language-specific implementations without general claims; blog posts and informal essays (except Abdelgawad 2016, included for completeness as most-cited informal argument)

Result: We reviewed the publications listed in the references under the inclusion criteria above; none satisfied the equivalence criteria defined below.

7.5.2 Equivalence Criteria We define five criteria that an "equivalent prior work" must satisfy:

Criterion	Definition	Why Required
Dominance theorem	Proves one discipline *strictly* dominates another (not just "trade-offs exist")	Core claim of this paper
Machine verification	Lean, Coq, Isabelle, Agda, or equivalent proof assistant with 0 incomplete proofs	Eliminates informal reasoning errors
Capability derivation	Capabilities derived from information structure, not enumerated	Proves completeness (no missing capabilities)
Impossibility proof	Proves structural typing *cannot* provide X (not just "doesn't")	Establishes necessity, not just sufficiency
Retrofit elimination	Proves adapters close the retrofit gap with bounded cost	Eliminates the "legacy code" exception

7.5.3 Prior Work Evaluation

Work	Dominance	Machine	Derived	Impossibility	Retrofit	Score
Cardelli [10]	—	—	—	—	—	0/5
Cook et al. [13]	—	—	—	—	—	0/5
Liskov & Wing [21]	—	—	—	—	—	0/5
Pierce [30]	—	—	—	—	—	0/5
Malayeri & Aldrich [22]	—	—	—	—	—	0/5
Gil & Maman [17]	—	—	—	—	—	0/5
Malayeri & Aldrich [24]	—	—	—	—	—	0/5
Abdelgawad & Cartwright [2]	—	—	—	—	—	0/5
Abdelgawad [3] (essay)	—	—	—	—	—	0/5
This paper	Thm 3.5	6300+ lines	Thm 3.43a	Thm 3.19	Thm 2.10j	**5/5**

Observation: In our survey, none of the works met any of the five criteria (all scored 0/5). To our knowledge, this paper is the first to satisfy all five.

7.5.4 Open Challenge

Open Challenge 7.1. Exhibit a publication satisfying *any* of the following:

1. Machine-checked proof (Lean/Coq/Isabelle/Agda) that nominal typing strictly dominates structural typing
2. Information-theoretic derivation showing the capability gap is complete (no missing capabilities)
3. Formal impossibility proof that structural typing cannot provide provenance, identity, enumeration, or conflict resolution
4. Proof that adapters eliminate the retrofit exception with O(1) cost
5. Decision procedure determining typing discipline from system properties

To our knowledge, no such publication exists. We welcome citations. The absence of any work scoring ≥1/5 in Table 7.5.3 is not a gap in our literature search. It reflects the state of the field.

7.5.5 Summary Table

Work	Contribution	What They Did NOT Prove	Our Extension
Malayeri & Aldrich [22, 24]	Qualitative trade-offs, empirical analysis	No formal proof of dominance	Strict dominance as formal theorem
Abdelgawad & Cartwright [2]	Inheritance = subtyping in nominal	No decision procedure	$B \neq \emptyset$ vs $B = \emptyset$ criterion

Work	Contribution	What They Did NOT Prove	Our Extension
Abdelgawad [3]	Information centralization (essay)	Not peer-reviewed, no machine proofs	Machine-checked Lean 4 formalization
Gil & Maman [17]	Whiteoak structural extension to Java	Hybrid justification, not dominance	Dominance when Bases axis exists
Veldhuizen [40]	Metaprogramming bounds	Type system specific	Cross-cutting application
Liskov & Wing [21]	Behavioral subtyping	Assumed nominal context	Field inheritance enforcement

The novelty gap in prior work. A comprehensive survey of 1988–2024 literature found: *"No single publication formally proves nominal typing strictly dominates structural typing when $B \neq \emptyset$."* Malayeri & Aldrich [22] observed trade-offs qualitatively; Abdelgawad [3] argued for nominal benefits in an essay; Gil & Maman [17] provided hybrid systems. None proved **strict dominance** as a theorem. None provided **machine-checked verification**. None **derived** the capability gap from information structure rather than enumerating it. None proved **adapters eliminate the retrofit exception** (Theorem 2.10j).

What we prove that prior work could not: 1. **Strict dominance as formal theorem** (Theorem 3.5): Nominal typing provides all capabilities of structural typing plus provenance, identity, enumeration at equivalent declaration cost. 2. **Information-theoretic completeness** (Theorem 3.19): The capability gap is *derived* from discarding the Bases axis, not enumerated. Any query distinguishing same-shape types requires B. This is mathematically necessary. 3. **Decision procedure** (Theorems 3.1, 3.4): $B \neq \emptyset$ vs $B = \emptyset$ determines which discipline is correct. This is decidable. 4. **Machine-checked proofs** (Section 6): 6300+ lines of Lean 4, 190+ theorems/lemmas, 0 `sorry` placeholders. 5. **Empirical validation at scale**: 13 case studies from a 45K LoC production system (OpenHCS).

Our core contribution: Prior work established that nominal and structural typing have trade-offs. We prove the trade-off is **asymmetric**: when $B \neq \emptyset$, nominal typing strictly dominates universally, not just in greenfield (Theorem 2.10j eliminates the retrofit exception). Duck typing is proven incoherent (Theorem 2.10d). Protocol is proven dominated (Theorem 2.10j). This follows necessarily from discarding the Bases axis.

Corollary 7.1 (Prior Work Comparison). A claim that these results were already established would need to exhibit a publication scoring $\geq 1/5$ in Table 7.5.3; we did not find one. If such a paper exists, we welcome a citation.

---

8 Discussion

8.1 Methodology and Disclosure

Role of LLMs in this work. This paper was developed through human-AI collaboration. The author provided the core intuitions, conjectures, and architectural insights; large language models (Claude, GPT-4) served as implementation partners—drafting proofs, suggesting formalizations, and generating code. The Lean 4 proofs were iteratively refined through this collaboration: the author specified what should be proved, the LLM proposed proof strategies, and the Lean compiler served as the ultimate arbiter of correctness.

This methodology aligns with the paper's thesis: the Lean proofs are *costly signals* (per the companion paper on credibility) because they require computational verification regardless of how they were generated. A proof that compiles is correct; the generation method is epistemically irrelevant to validity. The LLM accelerated exploration and drafting; the theorems stand or fall on their machine-checked proofs alone.

What the author contributed: The (B, S) decomposition, the strict dominance conjecture, the provenance impossibility claim, the connection to complexity bounds, the case study selection, and the architectural framing.

What LLMs contributed: LaTeX drafting, Lean tactic suggestions, literature search assistance, prose refinement, and exploration of proof strategies.

Why this disclosure matters: Academic norms around authorship and originality are evolving. We believe transparency about methodology strengthens rather than weakens the work. The proofs are machine-checked; the claims are falsifiable; the contribution is the insight, not the typing.

8.2 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.

Scope: systems where $B \neq \emptyset$. Simple scripts where the entire program fits in working memory may not require provenance tracking. But provenance is just one of four capabilities (Theorem 2.17). Even without provenance requirements, $\{B, S\}$ typing dominates because it provides identity, enumeration, and conflict resolution at no additional cost. Our theorems apply universally when $B \neq \emptyset$.

Remark: Traditional term: nominal typing.

Python as canonical model. The formalization uses Python's `type(name, bases, namespace)` because it is the clearest expression of the two-axis model. This is a strength, not a limitation: Python's explicit constructor exposes what other languages obscure with syntax. Table 2.2 demonstrates that 8 major languages (Java, C#, Rust, TypeScript, Kotlin, Swift, Scala, C++) are isomorphic to this model. Theorem 3.50 proves universality.

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.

Validation scope. The formal results (Theorems 3.5, 3.13, Corollary 6.3) are proven universally for any system where $B \neq \emptyset$. These proofs establish *what is impossible*: provenance cannot be computed without the bases axis (information-theoretically impossible, not merely difficult). The case studies (Section 5) demonstrate these theorems in a production codebase. The *direction* of the claims (that capability gaps translate to error reduction) follows from the formalism: if provenance is impossible without $\{B, S\}$ typing (Corollary 6.3), and provenance is required (PC = 1), then errors *must* occur under $\{S\}$-only incoherent typing. The *magnitude* of the effect is codebase-specific; the *existence* of the effect is not. We distinguish:

Remark: Traditional terms: nominal typing vs duck typing.

- **Universal (proven):** Capability gap exists, provenance is impossible under $\{S\}$-only typing, $\{B, S\}$ typing strictly dominates.
- **Singular (observed):** 47 `hasattr()` calls eliminated, centralized error detection via ABC contracts.

Remark: In type system terminology, "$\{S\}$-only typing" and "$\{B, S\}$ typing" correspond to duck/structural typing and nominal typing respectively.

We call for replication studies on other codebases to measure the magnitude of the effect across different architectural patterns. The formal results predict that *some* positive effect will be observed in any $B \neq \emptyset$ system requiring provenance; the specific multipliers are empirical questions.

OpenHCS is also wrong. The universal principle (Section 1.1) applies to OpenHCS itself. OpenHCS currently fixes the axis set $A = \{B, S, H\}$ (Bases, Shape, Hierarchy). By the Fixed Axis Incompleteness theorem, domains requiring axes outside this set (temporal versioning, provenance chains, security contexts, resource affinity) cannot be served.

OpenHCS is *less wrong* than $\{S\}$-only typing or $\{B, S\}$-only typing because it includes more axes ($A = \{B, S, H\}$). But "less wrong" is not "correct." The theorems predict that as the space of domains grows, OpenHCS will encounter impossibility walls:

$$\lim_{|\text{Domains}| \to \infty} |\{D : \neg \text{complete}(\{B, S, H\}, D)\}| = \infty$$

This is not speculation. It follows from `fixed_axis_incompleteness`: for each axis outside $\{B, S, H\}$, there exists a domain OpenHCS cannot serve. Since the space of possible axes is unbounded, so is the failure count.

Remark (Epistemic Strengthening). The incompleteness result does not depend on the axis space being *ontologically* infinite. Even if the space of possible axes were finite, no designer *knows* the complete set. The relevant constraint is epistemic: you cannot guarantee your fixed axis set is sufficient without proving completeness over all possible domains. Since no such proof exists for any fixed set, the designer's epistemic position is identical whether the axis space is infinite or merely unknown. This parallels Gödel's incompleteness (you cannot prove consistency from within) and Turing's halting problem (you cannot decide in advance which programs halt). The limitation is on what you can *know*, not merely on what *is*.

The path forward. Future versions of OpenHCS should parameterize over axis sets rather than hardcoding $\{B, S, H\}$. The architecture should be:

```
OpenHCS : AxisSet -> ConfigurationSystem
```

instantiated with $\{B, S, H\}$ for current use cases, but extensible to new axes without modification. This is the subject of ongoing work (see Section 8.4).

8.1 Axes as Query Dimensions

The paper's central claim—that axes are mandatory for specific capabilities—follows from a more primitive observation: *axes are query dimensions*. Every capability is operationally defined as answering a specific query, and each query requires traversal along specific axes.

Definition 8.1 (Query Parameterization). Let T denote type-level access to an object. Query complexity is parameterized by the axis set A required for resolution:

- T: Base object access (no axes required)
- $T(S)$: Attribute lookup—requires shape axis S
- $T(S, B)$: Attribute at inheritance level—requires shape S and bases B
- $T(S, B, H)$: Attribute at inheritance level within containment tree—requires S, B, and hierarchy H
- $T(S, B, H, \tau)$: Temporally-versioned attribute at inheritance level in containment tree—adds temporal axis τ

Theorem 8.1a (Query-Axis Necessity). A query of form $T(A)$ cannot be computed in a type system with axis set $A' \subsetneq A$. The query is information-theoretically impossible, not merely inefficient.

Proof. By construction, $T(A)$ requires traversal along dimension $a \in A \setminus A'$. Since a is absent from the type system, no structural path exists to resolve the query. Manual value-level encoding (e.g., `__source__` attributes) degrades the query to runtime inspection ($\Omega(n)$ error localization) but does not eliminate the axis—it encodes it in imperative state. ■

Corollary 8.1b (Provenance as $T(S, B)$ Query). "Which type provided this value?" is a $T(S, B)$ query: given field $f \in S$ (shape), determine $C \in \text{MRO}$ (bases) where f was defined. Duck typing ($\{S\}$-only) cannot answer this query (Corollary 6.3); nominal typing ($\{B, S\}$) can.

Example 8.1 (Hierarchy Query). OpenHCS's dual-axis resolver answers: "Given pipeline node n and field f, what is f's value in n's scope?" This is a $T(S, H)$ query (shape + hierarchy). Python provides $\{B, S\}$ natively; OpenHCS implements H manually via `ContextNode` graph traversal.

Theorem 8.1c (Manual Axis Complexity Superlinearity). Let A_{native} be a language's native axes and A_{req} be domain-required axes. For $k = |A_{\text{req}} \setminus A_{\text{native}}|$ manually-implemented axes, implementation complexity grows as $\Omega(k^2)$ due to synergistic interaction, not $O(k)$.

Proof sketch. Each manually-encoded axis a_i requires: (1) storage semantics (where axis data lives), (2) query interface (how to traverse the dimension), (3) consistency invariants (what axis combinations are valid). For k axes, pairwise interactions number $\binom{k}{2} = O(k^2)$. Examples of interactions:

- *Hierarchy* $\times$ *Bases:* When node n inherits configuration C, must hierarchy position propagate through MRO, or does MRO reset at each tree level?
- *Temporal* $\times$ *Hierarchy:* When pipeline structure changes over time, do version timestamps apply per-node or per-tree?
- *Security* $\times$ *Bases:* Do permission scopes inherit via MRO, or are they hierarchy-local?

Each interaction creates $O(1)$ design decisions, but incorrect choices lead to *irrecoverable architectural states*: local minima where refactoring requires rewriting all k axes simultaneously. This is the synergistic complexity penalty.

Native axes avoid this: the language runtime enforces interaction semantics (e.g., Python's MRO + descriptor protocol defines how $B \times S$ interact). Manual implementations must specify *all* interactions explicitly, and inconsistency across axes compounds error rates super-linearly. ■

Corollary 8.1d (Fixed-Axis Penalty Function). For a type system with native axes A_n and domain requiring A_d, the penalty for $|A_d \setminus A_n| = k$ missing axes is:

$$\text{Penalty}(k) = O(k) \text{ (storage)} + \Omega(k^2) \text{ (interactions)} + \text{LocalMinima}(k)$$

where LocalMinima(k) represents irrecoverable architectural debt from inconsistent interaction choices.

Remark. This formalizes why "just implement it manually" fails at scale. One missing axis ($k = 1$) is manageable. Two axes ($k = 2$) create $\binom{2}{2} = 1$ interaction. Three axes ($k = 3$) create $\binom{3}{2} = 3$ interactions. At $k = 5$ (e.g., temporal, security, provenance, resource affinity, versioning), $\binom{5}{2} = 10$ interactions must be specified manually, and the probability of reaching a local minimum approaches 1.

8.2.1 Connection to Degrees of Freedom

The super-linear complexity $\Omega(k^2)$ is not merely implementation cost—it directly corresponds to *degrees of freedom* (DOF) in the system architecture, as formalized in the companion paper on Single Source of Truth [4].

Coherence and Structural Facts. Paper 2 defines *coherence* as: all encoding locations agree on the same value for a fact. A *structural fact* is a fact about program structure (e.g., "Type T has method m" or "These are all types implementing interface I"). When DOF > 1, multiple independent locations can hold different values for the same structural fact, creating *incoherence*: the system cannot determine which value is "true." For manual axis implementation, each axis and each axis interaction is an independent encoding location, yielding DOF $= \Omega(k^2)$. This characterizes a *specific property*—coherence for structural facts—not general language quality. Languages make different trade-offs: static languages (Rust, Java, Go) choose compile-time safety over runtime introspection, accepting DOF > 1 as a consequence; dynamic languages (Python, CLOS, Smalltalk) enable DOF $= 1$ at the cost of performance and static guarantees.

Theorem 8.2 (DOF-Complexity Correspondence). For k manually-implemented axes, the architectural degrees of freedom satisfy:

$$\text{DOF}_{\text{total}} = k + \Omega(k^2)$$

where k represents storage DOF (one independent location per axis) and $\Omega(k^2)$ represents interaction DOF (independent decisions for each axis pair).

Proof. By Definition 2.6 of [4], DOF counts *independent encoding locations* for architectural facts. For k manually-implemented axes:

Storage DOF $= k$: Each axis a_i requires independent choice of:

- Where axis data is stored (object attribute, separate mapping, context variable)

- Data structure representation (dict, tree, graph)
- Mutability semantics (immutable, copy-on-write, in-place)

These k choices are *independent* in the sense that no language constraint forces agreement. Different axes can use different storage strategies, and the system remains type-correct (though architecturally inconsistent).

Interaction DOF $= \Omega(k^2)$: For each axis pair (a_i, a_j), independent choices include:

- *Traversal order:* Does query $T(a_i, a_j)$ traverse a_i first or a_j first?
- *Composition semantics:* Are values at (a_i, a_j) derived by combining separate $T(a_i)$ and $T(a_j)$ queries, or is the combination primitive?
- *Invalidation propagation:* When a_i changes, must a_j-indexed caches be invalidated?
- *Consistency invariants:* Can a_i and a_j hold contradictory information, or are they constrained to agree?

The number of axis pairs is $\binom{k}{2} = \frac{k(k-1)}{2} = \Omega(k^2)$. Each pair introduces $O(1)$ independent decisions. Therefore, interaction DOF $= O(k^2)$.

Total: $\text{DOF}_{\text{total}} = k + O(k^2) = \Omega(k^2)$. ■ ■

Corollary 8.2a (Modification Complexity Bound). By Theorem 6.2 of [4] (Non-SSOT Lower Bound), modification complexity for architectural changes satisfies:

$$M_{\text{effective}} = \Omega(\text{DOF}) = \Omega(k^2)$$

Proof. Theorem 6.2 of [4] establishes that for DOF $= n$ independent encoding locations, effective modification complexity $M_{\text{effective}} = \Omega(n)$. By Theorem 8.2, DOF $= \Omega(k^2)$ for k manually-implemented axes. Substituting: $M_{\text{effective}} = \Omega(k^2)$. ■ ■

Example 8.2 (OpenHCS Hierarchy Axis). OpenHCS's dual-axis configuration framework implements hierarchy H manually (Python provides B, S natively). With $k = 1$ manual axis:

- **Storage DOF $= 1$:** Hierarchy stored in `ContextNode` graph (~500 LOC)
- **Interaction DOF:**
  - $H \times B$: Does MRO traversal reset at each hierarchy level, or propagate through tree? (Decision: propagate)
  - $H \times S$: Are field accesses scoped per-node or per-subtree? (Decision: per-node with fallback)
- **Total:** DOF $= 1 + 2 = 3$ independent architectural decisions

For $k = 2$ axes (e.g., adding temporal versioning τ):

- **Storage DOF $= 2$:** Hierarchy + temporal storage
- **Interaction DOF:** $\binom{2}{2} + \binom{3}{2} = 1 + 3 = 4$ interactions ($H \times \tau$, plus 3 existing)
- **Total:** DOF $= 2 + 4 = 6$ (doubled from $k = 1$)

This quadratic growth explains why OpenHCS stopped at H—adding τ would require $\Omega(k^2)$ additional design decisions, each a potential inconsistency.

Remark. The DOF framework explains the "local minima" phenomenon: when k axes make inconsistent interaction choices, refactoring requires changing $\Omega(k^2)$ independent locations simultaneously. By Theorem 6.3 of [4] (Unbounded Gap), this creates modification complexity that grows without bound as the codebase scales. Native axes avoid this: the language enforces interaction semantics (e.g., Python's descriptor protocol defines $B \times S$ interaction), keeping DOF $= 1$ per axis.

8.1.1 Axiom Methodology

Theorem 8.1a (Axiom Scope). The axioms `Registry.wellFormed` and MRO monotonicity are *descriptive* of well-formed programs, not *restrictive* of the proof's scope. Programs violating these axioms are rejected by the language runtime before execution.

Proof. We enumerate each axiom and its enforcement:

Axiom	What It Requires	Language Enforcement
`Registry.wellFormed`	No duplicate ABC registrations, no cycles	`ABCMeta.register()` raises on duplicates; Python rejects cyclic inheritance
MRO monotonicity	If A <: B, A precedes B in MRO	C3 linearization guarantees this; violation raises `TypeError` at class definition
MRO totality	Every class has a linearizable MRO	C3 fails for unlinearizable diamonds; `TypeError` at class definition
`isinstance` correctness	`isinstance(x, T)` iff `type(x)` in T's subclass set	Definitional in Python's data model

A program violating any of these axioms fails at class definition time with `TypeError`. Such a program is not a valid Python program; it cannot be executed. Therefore, our theorems apply to *all valid programs.* ■

Corollary 8.1b (Axiom Scope). A claim that the axioms are too strong would require exhibiting: 1. A valid, executable Python program where the axioms fail, AND 2. A scenario where this program requires typing discipline analysis.

Programs where axioms fail are not valid programs; they crash at definition time. The axioms characterize well-formed programs, which is the standard scope for type system analysis.

Comparison to prior art. This methodology is standard in mechanized verification: - **CompCert** (verified C compiler): Assumes input is well-typed C - **seL4** (verified microkernel): Assumes hardware behaves according to spec - **CakeML** (verified ML compiler): Assumes input parses successfully

We follow the same pattern: assume the input is a valid program (accepted by Python's runtime), prove properties of that program. Proving that Python's parser and class system are correct is out of scope, and unnecessary, as Python's semantics are the *definition* of what we're modeling.

8.3 The Typing Discipline Hierarchy

Theorem 2.10d establishes that $\{S\}$-only incoherent typing (duck typing) is incoherent. Theorem 2.10g establishes that $\{S\}$-only typing with declared interfaces (structural typing) is eliminable when $B \neq \emptyset$. Together, these results collapse the space of valid typing disciplines.

The complete hierarchy:

Discipline	Coherent?	Eliminable?	When Valid
$\{S\}$-incoherent ($\{S\}$, incoherent)	No (Thm 2.10d)	N/A	Never
$\{S\}$-declared ($\{S\}$, coherent)	Yes	Yes, when $B \neq \emptyset$ (Thm 2.10g)	Only when $B = \emptyset$
$\{B, S\}$ ($\{B, S\}$, coherent)	Yes	No	Always (when $B \neq \emptyset$)

Remark: Traditional terms: Duck typing, Structural typing, Nominal typing.

$\{S\}$**-only incoherent typing** is incoherent: no declared interface, no complete compatibility predicate, no position on structure-semantics relationship. This is never valid.

Remark: Traditional term: duck typing.

$\{S\}$**-only declared typing (Protocol)** is coherent but eliminable: for any system using Protocol at boundaries, there exists an equivalent system using $\{B, S\}$ typing with explicit adapters (Theorem 2.10g). The only "value" of Protocol is avoiding the 2-line adapter class. Convenience is not a capability.

Remark: Traditional terms: structural typing vs nominal typing.

$\{B, S\}$ **typing (ABC)** is coherent and non-eliminable: it is the only necessary discipline for systems with inheritance.

Remark: Traditional term: nominal typing.

The eliminability argument. When integrating third-party type T that cannot inherit from your ABC:

```
# Structural approach (Protocol) - implicit
@runtime_checkable
class Configurable(Protocol):
    def validate(self) -> bool: ...

isinstance(their_obj, Configurable)  # Hope methods match

# Nominal approach (Adapter) - explicit
class TheirTypeAdapter(TheirType, ConfigurableABC):
    pass  # 2 lines. Now in your hierarchy.

adapted = TheirTypeAdapter(their_obj)  # Explicit boundary
isinstance(adapted, ConfigurableABC)   # Nominal check
```

The adapter approach is strictly more explicit. "Explicit is better than implicit" (Zen of Python). Protocol's only advantage (avoiding the adapter) is a convenience, not a typing capability.

Languages without inheritance. Go's struct types have $B = \emptyset$ by design. $\{S\}$-only declared typing with declared interfaces is the only coherent option. Go does not use $\{S\}$-only incoherent typing; Go interfaces are declared [37]. This is why Go's type system is sound despite lacking inheritance.

Remark: Traditional terms: structural typing, duck typing.

The final collapse. For languages with inheritance ($B \neq \emptyset$): - $\{S\}$-only incoherent typing: incoherent, never valid - $\{S\}$-only declared typing: coherent but eliminable, valid only as convenience - $\{B, S\}$ typing: coherent and necessary

Remark: Traditional terms: duck typing, structural typing, nominal typing.

The only *necessary* typing discipline is $\{B, S\}$. Everything else is either incoherent ($\{S\}$-only incoherent typing) or reducible to $\{B, S\}$ with trivial adapters ($\{S\}$-only declared typing).

Remark: Traditional terms: nominal, duck typing, structural typing.

8.4 Future Work

Gradual $\{B, S\}/\{S\}$-declared typing. TypeScript supports both $\{B, S\}$ (via branding) and $\{S\}$-only declared typing in the same program. Formalizing the interaction between these disciplines, and proving soundness of gradual migration, would enable principled adoption strategies.

Remark: Traditional terms: nominal, structural typing.

Trait systems. Rust traits and Scala traits 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 Ω(n)? Such a tool would help language designers evaluate typing discipline tradeoffs during language design.

Axis-parameterized type systems. The universal principle (Section 1.1) establishes that all fixed-axis systems are incomplete. The correct architecture is $L : \text{AxisSet} \to \text{TypeSystem}$, where axes are injected rather than hardcoded. Future work includes:

1. **Formalizing axis-parameterized type systems in Lean 4.** The `axis_framework.lean` file proves the Fixed Axis Incompleteness and Parameterized Immunity theorems. Extending this to a full type system formalization would provide machine-checked guarantees for domain-agnostic configuration systems.

2. **OpenHCS parameterization.** Refactoring OpenHCS from fixed $\{B, S, H\}$ to parameterized axis injection. The architecture would become `OpenHCS : AxisSet -> ConfigurationSystem`, instantiated with $\{B, S, H\}$ for current domains but extensible to temporal versioning, security contexts, or resource affinity axes without framework modification.

3. **Python `type()` extension (PEP proposal).** Python's class constructor `type(name, bases, namespace)` hardcodes a fixed axis set: B (bases) and S (namespace). This is a fixed-axis design. We propose extending the signature to:

   ```
   type(name, bases, namespace, **axes)
   ```

   This would allow axis injection at type creation time:

   ```
   # Current: fixed {B, S}
   Foo = type("Foo", (Bar,), {"x": 1})

   # Proposed: parameterized axes
   Foo = type("Foo", (Bar,), {"x": 1},
              hierarchy=ctx,         # H axis: containment position
              version=v,             # Temporal axis
              provenance=chain)      # Provenance axis
   ```

   The `**axes` parameter would be accessible via `cls.__axes__` and introspectable at runtime. This enables: (a) domain-specific axis requirements declared at the type level, (b) framework code parameterized over arbitrary axes, and (c) gradual adoption; existing code uses the 3-argument form unchanged.

   A formal PEP would specify: (i) storage semantics (`__axes__` dict), (ii) inheritance behavior (axis merging vs. override), (iii) metaclass interaction, and (iv) static type checker support (`typing.Axes`). The mathematical foundation is provided by the Fixed Axis Incompleteness theorem: Python's current `type()` will fail for domains requiring axes outside $\{B, S\}$. Parameterization eliminates this impossibility wall.

8.5 Implications for Language Design

Language designers face a fundamental choice: provide nominal typing (enabling provenance), structural typing (for $B = \emptyset$ boundaries), 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 the relationship: when $B \neq \emptyset$, $\{B, S\}$ typing (ABCs) strictly dominates Protocol (Theorem 2.10j). Protocol provides convenience (avoiding adapters) but this is not a capability; ABCs can also integrate external types via adapters. Protocol is dominated: it provides a strict subset of capabilities.

Remark: In Python, ABCs implement $\{B, S\}$ typing (nominal), while Protocols implement $\{S\}$-only typing (structural).

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` makes $\{B, S\}$ typing strictly optimal. If a language exposes explicit inheritance declarations (`class C(Base)`), Theorem 3.4 (Axis Inclusion Dominance) applies: $\{B, S\}$ typing strictly dominates $\{S\}$-only typing. Language designers cannot add inheritance to a structurally-typed language without creating capability gaps that $\{B, S\}$ typing would eliminate.

Remark: In type system terminology, $\{B, S\}$ and $\{S\}$ correspond to nominal and structural typing respectively.

8.6 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 / KLOC
```

Measures ad-hoc capability probing. High DTD where $B \neq \emptyset$ indicates discipline violation. We count only `hasattr()`, not `getattr()` or `try/except AttributeError`, because `hasattr()` is specifically capability detection ("does this object have this attribute?"), the operational signature of $\{S\}$-only typing (Definition 2.10c). `getattr()` without a fallback is explicit attribute access; `getattr()` with a fallback or `try/except AttributeError` may indicate $\{S\}$-only typing but also appear in legitimate metaprogramming (descriptors, `__getattr__` hooks, optional feature detection at system boundaries). The theorem backing (Theorem 2.10d) establishes `hasattr()` as the incoherent probe; other patterns require case-by-case analysis.

Remark: "Duck typing" is traditional terminology for $\{S\}$-only typing.

Metric 2: $\{B, S\}$ Typing Ratio (BTR)

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

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

Remark: Previously called "Nominal Typing Ratio (NTR)". In type system terminology, $\{B, S\}$ typing corresponds to nominal typing.

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 Ω(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 any code where $B \neq \emptyset$ (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).

8.7 Hybrid Systems and Methodology Scope

Our theorems establish necessary conditions for provenance-tracking systems. This section clarifies the relationship between dominance and practical constraints; shape-based typing is never an alternative, only a sacrifice forced by external constraints.

8.6.1 Structural Typing Is Eliminable (Theorem 2.10g) **Critical update:** Per Theorem 2.10g, structural typing is *eliminable* when $B \neq \emptyset$. The scenarios below describe when Protocol is *convenient*, not when it is *necessary*. In all cases, the explicit adapter approach (Section 8.2) is available and strictly more explicit.

Retrofit scenarios. When integrating independently developed components that share no common base classes, you cannot mandate inheritance directly. However, you *can* wrap at the boundary: `class TheirTypeAdapter(TheirType, YourABC): pass`. Protocol is a convenience that avoids this 2-line adapter. Duck typing is never acceptable.

Language boundaries. Calling from Python into C libraries, where inheritance relationships are unavailable. The C struct has no `bases` axis. You can still wrap at ingestion: create a Python adapter class that inherits from your ABC and delegates to the C struct. Protocol avoids this wrapper but does not provide capabilities the wrapper lacks.

Versioning and compatibility. When newer code must accept older types that predate a base class introduction, you can create versioned adapters: `class V1ConfigAdapter(V1Config, ConfigBaseV2): pass`. Protocol avoids this but does not provide additional capabilities.

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 (effectively $B = \emptyset$ at runtime), structural typing has no capability gap because there's no B to lose. However, see Section 8.7 for why TypeScript's *class-based* structural typing creates tension; once you have `class extends`, you have $B \neq \emptyset$.

Summary. In all scenarios with $B \neq \emptyset$, the adapter approach is available. Protocol's only advantage is avoiding the adapter. Avoiding the adapter is a convenience, not a typing capability (Corollary 2.10h).

8.6.2 The $B \neq \emptyset$ vs $B = \emptyset$ Criterion The only relevant question is whether inheritance exists:

$B \neq \emptyset$ **(inheritance exists):** Nominal typing is correct. Adapters handle external types (Theorem 2.10j). Examples: - OpenHCS config hierarchy: `class PathPlanningConfig(GlobalConfigBase)` - External library types: wrap with `class TheirTypeAdapter(TheirType, YourABC): pass`

$B = \emptyset$ **(no inheritance):** Structural typing is the only option. Examples: - JSON objects from external APIs - Go interfaces - C structs via FFI

The "greenfield vs retrofit" framing is obsolete (see Remark after Theorem 3.62).

8.6.3 System Boundaries Systems have $B \neq \emptyset$ components (internal hierarchies) and $B = \emptyset$ boundaries (external data):

```
# B != {}: internal config hierarchy (use nominal)
class ConfigBase(ABC):
    @abstractmethod
    def validate(self) -> bool: pass

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

# B = {}: parse external JSON (structural is only option)
def load_config_from_json(json_dict: Dict[str, Any]) -> ConfigBase:
    # JSON has no inheritance; structural validation at boundary
    if "well_filter" in json_dict:
        return PathPlanningConfig(**json_dict)  # Returns nominal type
    raise ValueError("Invalid config")
```

The JSON parsing layer is $B = \emptyset$ (JSON has no inheritance). The return value is $B \neq \emptyset$ (ConfigBase hierarchy). This is correct: structural at data boundaries where $B = \emptyset$, nominal everywhere else.

8.6.4 Scope Summary

Context	Typing Discipline	Justification
$B \neq \emptyset$ (any language with inheritance)	Nominal (mandatory)	Theorem 2.18 (strict dominance), Theorem 2.10j (adapters dominate Protocol)
$B = \emptyset$ (Go, JSON, pure structs)	Structural (correct)	Theorem 3.1 (namespace-only)
Language boundaries (C/FFI)	Structural (mandatory)	No inheritance available ($B = \emptyset$ at boundary)

Removed rows: - "Retrofit / external types → Structural (acceptable)". Adapters exist (Theorem 2.10j); structural is dominated. "Small scripts / prototypes → Duck (acceptable)". Duck typing is incoherent for B-dependent queries (Theorem 2.10d).

The methodology states: **if $B \neq \emptyset$, nominal typing is the capability-maximizing choice.** Protocol is dominated. Duck typing is incoherent. The decision follows from the capability analysis, not from project size or aesthetic preference.

8.8 Case Study: TypeScript's Design Tension

TypeScript presents a puzzle: it has explicit inheritance (`class B extends A`) but uses structural subtyping. Is this a valid design tradeoff, or an architectural tension with measurable consequences? The runtime model (JavaScript prototypes) preserves B and nominal identity (via `instanceof`), while the static checker erases B when computing compatibility [38, 6]. Per Definition 8.3 this is incoherence.

Definition 8.3 (Type System Coherence). A type system is *coherent* with respect to a language construct if the type system's judgments align with the construct's runtime semantics. Formally: if construct C creates a runtime distinction between entities A and B, a coherent type system also distinguishes A and B.

Definition 8.4 (Type System Tension). A type system exhibits *tension* when it is incoherent (per Definition 8.3) AND users create workarounds to restore the missing distinctions.

8.7.1 The Tension Analysis TypeScript's design exhibits three measurable tensions:

Tension 1: Incoherence per Definition 8.3.

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

// Runtime: instanceof creates distinction
const b = new B();
console.log(b instanceof A);  // false - different classes

// Type system: no distinction
function f(a: A) { }
f(new B());  // OK - same structure
```

The `class` keyword creates a runtime distinction (`instanceof` returns `false`). The type system does not reflect this distinction. Per Definition 8.3, this is incoherence: the construct (`class`) creates a runtime distinction that the type system ignores.

Tension 2: Workaround existence per Definition 8.4.

TypeScript programmers use "branding" to restore 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)
```

The existence of this workaround demonstrates Definition 8.4: users create patterns to restore distinctions the type system fails to provide. TypeScript GitHub issue #202 (2014) and PR #33038 (2019) request or experiment with native nominal types [26, 27], confirming the workaround is widespread.

Tension 3: Measurable consequence.

The `extends` keyword is provided but ignored by the type checker. This is information-theoretically suboptimal per our framework: the programmer declares a distinction (`extends`), the type system discards it, then the programmer re-introduces a synthetic distinction (`__brand`). The same information is encoded twice with different mechanisms.

8.7.2 Formal Characterization **Theorem 8.7 (TypeScript Incoherence).** TypeScript's class-based type system is incoherent per Definition 8.3.

Proof. 1. TypeScript's `class A` creates a runtime entity with nominal identity (JavaScript prototype) 2. `instanceof A` checks this nominal identity at runtime 3. TypeScript's type system uses structural compatibility for class types 4. Therefore: runtime distinguishes `A` from structurally-identical `B`; type system does not 5. Per Definition 8.3, this is incoherence. ■

Corollary 8.7.1 (Branding Validates Tension). The prevalence of branding patterns in TypeScript codebases empirically validates the tension per Definition 8.4.

Evidence. TypeScript GitHub issue #202 (2014, 1,200+ reactions) and PR #33038 (2019) request native nominal types [26, 27]. The `@types` ecosystem includes branded type utilities (`ts-brand`, `io-ts`). This is observed community behavior consistent with the predicted tension.

8.7.3 Implications for Language Design TypeScript's tension is an intentional design decision for JavaScript interoperability. The structural type system allows gradual adoption in untyped JavaScript codebases. However, TypeScript has `class` with `extends`, meaning $B \neq \emptyset$. Our theorems apply: $\{B, S\}$ typing strictly dominates (Theorem 3.5).

The tension manifests in practice: programmers use `class` expecting $\{B, S\}$ semantics, receive $\{S\}$-only semantics, then add branding to restore $\{B, S\}$ behavior. Our theorems predict this: Theorem 3.4 shows that when `bases` exist, $\{B, S\}$ typing strictly dominates $\{S\}$-only typing; TypeScript violates this optimality, causing measurable friction. The branding idiom is programmers manually recovering capabilities the language architecture foreclosed.

Remark: In type system terminology, $\{B, S\}$ and $\{S\}$ correspond to nominal and structural typing.

The lesson: Languages adding `class` syntax should consider whether their type system will be coherent (per Definition 8.3) with the runtime semantics of class identity. $\{S\}$-only typing is correct for languages without inheritance (Go). For languages with inheritance, coherence requires $\{B, S\}$ typing or explicit documentation of the intentional tension.

8.9 Mixins with MRO Strictly Dominate Object Composition

The "composition over inheritance" principle from the Gang of Four [16] 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 -> Logging -> Caching -> 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] Multiple behaviors combined

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

Addressing runtime swapping: A common objection is that composition allows "swapping implementations at runtime" (`handler.cache = NewCache()`). This is orthogonal to the dominance claim for two reasons:

1. **Mixins can also swap at runtime** via class mutation: `Handler.__bases__ = (NewLoggingMixin, CachingMixin, BaseHandler)` or via `type()` to create a new class dynamically. Python's class system is mutable.

2. **Runtime swapping is a separate axis.** The dominance claim concerns *static behavior extension*: adding logging, caching, validation to a class. Whether to also support runtime reconfiguration is an orthogonal requirement. Systems requiring runtime swapping can use mixins for static extension AND composition for swappable components. The two patterns are not mutually exclusive.

Therefore: **Mixin capabilities** $\supset$ **Composition capabilities** (strict superset) for static behavior extension.

Theorem 8.1 (Mixin Dominance). For static 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. ■

Corollary 8.1.1 (Runtime Swapping Is Orthogonal). Runtime implementation swapping is achievable under both patterns: via object attribute assignment (composition) or via class mutation/dynamic type creation (mixins). Neither pattern forecloses this capability.

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) $\subset$ capabilities(D_B) (strict subset). ■

8.10 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 Ω(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: when $B \neq \emptyset$, use nominal typing.

Historical validation: Python's evolution confirms our theorems. Python 1.0 (1991) had only duck typing, an incoherent non-discipline (Theorem 2.10d). Python 2.6 (2007) added ABCs because duck typing was insufficient for large codebases. Python 3.8 (2019) added Protocols for retrofit scenarios; coherent structural typing to replace incoherent duck typing. This evolution from incoherent → nominal → nominal+structural exactly matches our formal predictions.

8.11 Connection to Gradual Typing

Our results connect to the gradual typing literature (Siek & Taha 2006, Wadler & Findler 2009). Gradual typing addresses adding types to existing untyped code. Our theorems address which discipline to use when $B \neq \emptyset$.

The complementary relationship:

Scenario	Gradual Typing	Our Theorems
Untyped code ($B = \emptyset$)	[PASS] Applicable	[N/A] No inheritance
Typed code ($B \neq \emptyset$)	[N/A] Already typed	[PASS] Nominal dominates

Gradual typing's insight: When adding types to untyped code, the dynamic type `?` allows gradual migration. This applies when $B = \emptyset$ (no inheritance structure exists yet).

Our insight: When $B \neq \emptyset$, nominal typing strictly dominates. This includes "retrofit" scenarios with external types; adapters make nominal typing available (Theorem 2.10j).

The unified view: Gradual typing and nominal typing address orthogonal concerns: - Gradual typing: Typed vs untyped ($B = \emptyset \to B \neq \emptyset$ migration) - Our theorems: Which discipline when $B \neq \emptyset$ (answer: nominal)

Theorem 8.3 (Gradual-Nominal Complementarity). Gradual typing and nominal typing are complementary, not competing. Gradual typing addresses the presence of types; our theorems address which types to use.

Proof. Gradual typing's dynamic type `?` allows structural compatibility with untyped code where $B = \emptyset$. Once $B \neq \emptyset$ (inheritance exists), our theorems apply: nominal typing strictly dominates (Theorem 3.5), and adapters eliminate the retrofit exception (Theorem 2.10j). The two address different questions. ∎

8.12 Connection to Leverage Framework

The strict dominance of nominal typing (Theorem 2.10j) is an instance of a more general principle: *leverage maximization*.

Define **leverage** as $L = |\text{Capabilities}|/\text{DOF}$, where DOF (Degrees of Freedom) counts independent encoding locations for type information. Both typing disciplines have similar DOF (both require type declarations at use sites), but nominal typing provides 4 additional capabilities (provenance, identity, enumeration, conflict resolution). Therefore:

$$L(\text{nominal}) = \frac{5}{1} > \frac{1}{1} = L(\text{duck})$$

The leverage framework (see companion paper) proves that for any architectural decision, the optimal choice maximizes leverage. This paper proves the *instance*; the companion paper proves the *metatheorem* that leverage maximization is universally optimal.

Theorem 8.4 (Typing as Leverage Instance). The strict dominance of nominal typing (Theorem 2.10j) is an instance of the Leverage Maximization Principle.

Proof. By Theorem 2.10j, nominal typing provides a strict superset of capabilities at equivalent cost. This is exactly the condition for higher leverage: $L(\text{nominal}) > L(\text{duck})$. By the Leverage Maximization Principle, nominal typing is therefore optimal. ∎

9 Conclusion

9.1 The Universal Result

This paper proves an impossibility theorem: **any classification system with fixed axes is incomplete for some domain.** This is not a limitation of specific implementations; it is information-theoretic. The data required to answer certain queries does not exist in fixed-axis systems.

The corollary is equally strong: **axis-parameterized systems dominate fixed systems absolutely.** A parameterized system can instantiate any axis set, achieving completeness for any domain. The asymmetry is:

- Fixed: $\forall A. \exists D. \neg\text{complete}(A, D)$. Every fixed system fails somewhere
- Parameterized: $\forall D. \exists A. \text{complete}(A, D)$. Every domain is reachable

This result applies to any classification scheme: type systems, ontologies, taxonomies, database schemas, knowledge graphs. The type system instantiation is one case study.

9.2 The Type System Instantiation

For programming languages with class-based inheritance:

1. **The $B = \emptyset$ criterion**: If a language has inheritance ($B \neq \emptyset$), including B in the axis set is required for capability-maximization (Theorem 2.18). If a language lacks inheritance ($B = \emptyset$), $\{S\}$-only systems (structural typing) are correct. Duck typing is incoherent in both cases (Theorem 2.10d). **Terminology note:** In type system terminology, $\{B, S\}$-inclusive systems are called "nominal typing"; $\{S\}$-only systems are called "structural typing."
2. **Complexity separation**: Systems using $\{B, S\}$ achieve O(1) error localization versus $\{S\}$-only incoherent use (duck typing) requiring $\Omega(n)$ (Theorem 4.3). The gap is unbounded.
3. **Provenance impossibility**: Systems using only $\{S\}$ cannot provide provenance (Corollary 6.3, machine-checked). This is information-theoretic, not implementational.
4. **Hierarchical extension**: For systems with scope hierarchies, (B, S, H) is the unique minimal complete axis set (Theorem 3.63).

The decision procedure: Given domain requirements, the required axes are *computable* (Theorem 2.26). The typing discipline follows deterministically.

9.2.1 Summary of Results

The decision procedure (Theorem 3.62) outputs "$\{B, S\}$ typing" when $B \neq \emptyset$ and "$\{S\}$-only typing" when $B = \emptyset$. All proofs are machine-checked (Lean 4, 0 `sorry`).

Remark: Traditional terms: nominal typing, structural typing.

Two architects examining identical requirements will derive identical discipline choices. Disagreement indicates incomplete requirements or different analysis; the formal framework provides a basis for resolution.

Incoherence of denial. The uniqueness theorems establish $\neg\exists$ alternatives to the minimal complete axis set. The position "these results are interesting but classification scheme design remains a preference" presupposes $\exists$ alternatives. Accepting the theorems while maintaining preference instantiates $P \wedge \neg P$ (logical incoherence, not mere disagreement).

For type systems specifically: this work does not contribute to the debate over typing disciplines. It **dissolves** it by showing the debate presupposes a choice that doesn't exist. Axes are derived from domain requirements (Theorem 2.26); typing discipline names ("nominal," "structural") are labels for specific axis configurations. For classification systems generally: the "which taxonomy should we use" question has a mathematical answer given the domain.

On capability vs. aesthetics. Aesthetics, elegance, and readability are orthogonal to capability. The theorems establish that $\{B, S\}$ typing provides strictly more capabilities. This is a mathematical fact, not a stylistic preference. Choosing fewer capabilities is a sacrifice, not an alternative. If constraints force that sacrifice (e.g., interoperability with systems lacking type metadata), the sacrifice is justified, but it remains a sacrifice.

Remark: Traditional term: nominal typing.

On PEP 20 (The Zen of Python). PEP 20 is sometimes cited to justify $\{S\}$-only incoherent typing. However, several Zen principles align with $\{B, S\}$ typing: "Explicit is better than implicit" (ABCs are explicit; hasattr is implicit), and "In the face of ambiguity, refuse the temptation to guess" ($\{S\}$-only incoherent typing infers interface conformance; $\{B, S\}$ typing verifies it). We discuss this alignment in Section 8.9.

Remark: Traditional terms: duck typing, nominal typing.

9.3 Application: LLM Code Generation

The decision procedure (Theorem 3.62) has a clean application domain: evaluating LLM-generated code.

Why LLM generation is a clean test. When a human prompts an LLM to generate code, the $B \neq \emptyset$ vs $B = \emptyset$ distinction is explicit in the prompt. "Implement a class hierarchy for X" has $B \neq \emptyset$. "Parse this JSON schema" has $B = \emptyset$. Unlike historical codebases, which contain legacy patterns, metaprogramming artifacts, and accumulated technical debt, LLM-generated code represents a fresh choice about typing discipline.

Corollary 9.1 (LLM Discipline Evaluation). Given an LLM prompt with explicit context: 1. If the prompt involves inheritance ($B \neq \emptyset$) $\rightarrow$ isinstance/ABC patterns are correct; hasattr patterns are violations (by Theorem 3.5) 2. If the prompt involves pure data without inheritance ($B = \emptyset$, e.g., JSON) $\rightarrow$ structural patterns are the only option 3. External types requiring integration $\rightarrow$ use adapters to achieve nominal (Theorem 2.10j) 4. Deviation from these patterns is a typing discipline error detectable by the decision procedure

Proof. Direct application of Theorem 3.62. The generated code's patterns map to discipline choice. The decision procedure evaluates correctness based on whether $B \neq \emptyset$. ∎

Implications. An automated linter applying our decision procedure could: - Flag `hasattr()` in code with inheritance as a discipline violation - Suggest `isinstance()`/ABC replacements - Validate that provenance-requiring prompts produce nominal patterns - Flag Protocol usage as dominated (Theorem 2.10j)

This application is clean because the context is unambiguous: the prompt explicitly states whether the developer controls the type hierarchy. The metrics defined in Section 8.5 (DTD, NTR) can be computed on generated code to evaluate discipline adherence.

Falsifiability. If code with $B \neq \emptyset$ consistently performs better with $\{S\}$-only patterns than $\{B, S\}$ patterns, our Theorem 3.5 is falsified. We predict it will not.

Remark: Traditional terms: structural patterns vs nominal patterns.

9.4 Data Availability

OpenHCS Codebase: The OpenHCS platform (45K LoC Python) is available at https://github.com/trissim/openhcs [35]. The codebase demonstrates the practical application of the theoretical framework, including the hierarchical scoping system (H axis) and ABC-based contracts.

PR #44: The migration from $\{S\}$-only incoherent typing to $\{B, S\}$ contracts is documented in a publicly verifiable pull request [36]: https://github.com/trissim/openhcs/pull/44. This PR eliminated 47 scattered `hasattr()` checks by introducing ABC contracts.

Remark: Traditional terms: duck typing to nominal contracts.

Lean 4 Proofs: The complete Lean 4 formalization (6300+ lines, 190+ theorems, 0 `sorry` placeholders) [34] is included as supplementary material. Reviewers can verify the proofs by running `lake build` in the proof directory.

Reproducibility: Install OpenHCS via `pip install openhcs` to observe the H-axis behaviors described in Section 5 (click-to-provenance navigation, flash propagation).

A Completeness and Robustness Analysis

This appendix provides detailed analysis addressing potential concerns about the scope, applicability, and completeness of our results.

A.1 Comprehensive Concern Analysis

We identify the major categories of potential concerns and demonstrate why each does not affect our conclusions.

Potential Concern	Formal Analysis
"Model is incomplete"	Theorem 3.32 (Model Completeness)
"Duck typing has tradeoffs"	Theorems 3.34-3.36 (Capability Comparison)
"Axioms are assumptive"	Lemma 3.37 (Axiom is Definitional)
"Clever extension could fix it"	Theorem 3.39 (Extension Impossibility)
"What about generics?"	Theorems 3.43-3.48, Table 2.2 (Parameterized N)

Potential Concern	Formal Analysis
"Erasure changes things"	Theorems 3.46-3.47 (Compile-Time Type Checking)
"Only works for some languages"	Theorem 3.47 (8 languages), Remark 3.49 (exotic features)
"What about intersection/union types?"	Remark 3.49 (still two axes)
"What about row polymorphism?"	Remark 3.49 (pure S, loses capabilities)
"What about higher-kinded types?"	Remark 3.49 (parameterized N)
"Only applies to greenfield"	Theorem 2.10j (Adapters eliminate retrofit exception)
"Legacy codebases are different"	Corollary 3.51 (sacrifice, not alternative)
"Claims are too broad"	Theorem 2.6 (axis requirements are domain-derived)
"Dominance $\neq$ migration"	Theorem 3.55 (Dominance $\neq$ Migration)
"Greenfield is undefined"	Definitions 3.57-3.58, Theorem 3.59
"Provenance requirement is circular"	Theorem 3.61 (Provenance Detection)
"You just need discipline"	Concern 8: Discipline is the external oracle
"The proofs are trivial"	Concern 9: Definitional proofs from correct modeling
"Java/Rust can do this too"	Concern 10: Enable vs enforce distinction
"This is just subtyping"	Concern 11: Construction vs relationship (orthogonal layers)

A.2 Detailed Analysis of Each Concern

We expand the most common concerns below; the remaining items in the table above are direct corollaries of the referenced results.

Concern 1: Model Completeness. *Potential concern:* The (B, S) model may fail to capture relevant aspects of type systems.

Analysis: Theorem 3.5 establishes model completeness by constitutive definition. In Python, `type(name, bases, namespace)` is the universal type constructor. A type does not merely *have* (B, S); a type *is* (B, S). Any computable function over types is therefore definitionally a function of this pair. Properties like `__mro__` or `__module__` are not counterexamples: they are derived from or stored within (B, S). The syntactic name (`__name__`) is metadata stored in the namespace (S axis), not an independent axis. This is definitional closure, not empirical enumeration. No third axis can exist because the pair is constitutive.

Concern 2: Duck Typing Tradeoffs. *Potential concern:* Duck typing has flexibility that nominal typing lacks.

Analysis: Theorems 3.34-3.36 establish that nominal typing provides a strict superset of duck typing capabilities. Duck typing's "acceptance" of structurally-equivalent types is not a capability: it is the *absence* of the capability to distinguish them. We treat "capability" as the set of definable operations/predicates available to the system, not the cost of retrofitting legacy code; migration/retrofit cost is handled separately (Theorem 3.5, adapter results in Theorem 2.4).

Concern 3: Axiom Circularity. *Potential concern:* The axioms are chosen to guarantee the conclusion.

Analysis: Lemma 3.5 establishes that the axiom "shape-based typing treats same-namespace types identically" is not an assumption: it is the *definition* of shape-based typing (Definition 2.10).

Concern 4: Future Extensions. *Potential concern:* A clever extension to duck typing could recover provenance.

Analysis: Theorem 3.5 proves that any computable extension over $\{S\}$ (even with the leaf of B, i.e., type names) cannot recover provenance. The limitation is structural, not technical. A common response is "just check `type(x)`", but this proves the point: inspecting `type(x)` gives you the leaf of B. To get provenance, you need the full lineage. Once you consult the full B axis, you have left shape-only typing and moved to nominal typing. The "fix" is the adoption of our thesis.

Concern 5: Generics and Parametric Polymorphism. *Potential concern:* The model doesn't handle generics.

Analysis: Theorems 3.43-3.48 establish that generics preserve the axis structure. Type parameters are a refinement of N, not additional information orthogonal to (B, S).

Concern 6: Single Codebase Evidence. *Potential concern:* Evidence is from one codebase (OpenHCS).

Analysis: This objection conflates **existential witnesses** with **premises**. A category error. In logic, a premise is something the conclusion depends on; an existential witness demonstrates satisfiability.

The dominance theorems are proven from the *definition* of shape-based typing (Lemma 3.5: the axiom is definitional). Examine the proof of Theorem 3.2 (Provenance Impossibility): it proceeds by showing that (S) contains insufficient information to compute provenance. This is an information-theoretic argument that references no codebase. You could prove this theorem before any codebase existed.

OpenHCS appears only to demonstrate that the four capabilities are *achievable*. That a real system uses provenance, identity, enumeration, and conflict resolution. This is an existence proof ("such systems exist"), not a premise ("if OpenHCS works, then the theorems hold").

Analogy: Proving "comparison-based sorting requires $\Omega(n \log n)$ comparisons" does not require testing on multiple arrays. The proof is structural. Exhibiting quicksort demonstrates the bound is achievable, not that the theorem is true. Similarly, our theorems follow from (B, S) structure; OpenHCS demonstrates achievability.

Concern 7: Scope Confusion. *Potential concern:* Discipline dominance implies migration recommendation.

Analysis: Theorem 3.5 formally proves that Pareto dominance of discipline A over B does NOT imply that migrating from B to A is beneficial for all codebases. Dominance is codebase-independent; migration cost is codebase-dependent.

Concern 8: You Just Need Discipline. *Potential concern:* Real teams maintain consistency through code review, documentation, and discipline. Language features are unnecessary.

Analysis: Discipline *is* the external oracle. When multiple locations encode the same fact (DOF > 1), consistency requires an oracle to resolve disagreements. "Code review and documentation" are exactly that oracle—human-maintained, fallible, bypassable.

The question is whether the oracle is:

- **Internal** (language-enforced, automatic, unforgeable), or

- **External** (human-maintained, fallible, bypassable)

Nominal typing with Python's `__subclasses__()` provides an internal oracle: the language guarantees the registry is complete. Duck typing requires an external oracle: humans must remember to update all locations. Both achieve consistency when they work. The difference is failure mode: language enforcement cannot be forgotten; human discipline can.

This is not a counterargument to our thesis; it is the thesis restated. We prove that systems without nominal typing require external oracles.

Concern 9: The Proofs Are Trivial. *Potential concern:* Most proofs are just `rfl` (reflexivity). They're trivial tautologies, not real theorems.

Analysis: When modeling is correct, theorems become definitional. This is a feature, not a bug.

Consider: "The sum of two even numbers is even." In a well-designed formalization, this might be `rfl`—not because it's trivial, but because the definition of "even" makes the property structural.

Not all proofs are `rfl`. The `provenance_impossibility` theorem requires 40+ lines:

1. Assume a hypothetical provenance function exists for $\{S\}$-only systems
2. Construct two types with identical S but different B (different lineages)
3. Show the function must return different values for indistinguishable inputs
4. Contradiction

The proof structure (assumption $\to$ construction $\to$ contradiction) is genuine mathematical reasoning. The `rfl` proofs establish scaffolding; substantive proofs build on that scaffolding.

Concern 10: Static Languages Achieve This With Reflection/Macros. *Potential concern:* Java has reflection. Rust has proc macros. These languages can achieve the same capabilities.

Analysis: Reflection and macros enable *patterns* but do not *enforce* them. The critical distinction:

Java reflection: You can query `Class.getDeclaredMethods()` at runtime, but:

- No automatic hook fires when a class is defined
- You must manually maintain registries (DOF ≥ 2)
- The pattern is bypassable: create a class without registering it

Rust proc macros: You can generate code at compile time, but:

- Macros are per-item isolated—they cannot see other items during expansion
- Registration is bypassable: `impl Trait` without `#[derive]` annotation
- The `inventory` crate uses linker tricks external to language semantics

Contrast Python:

```
class Handler(Registry):  # __init_subclass__ fires AUTOMATICALLY
    pass  # Cannot create unregistered subclass---IMPOSSIBLE
```

Python's hook is *unforgeable*. The language semantics guarantee that creating a subclass triggers the hook. There is no syntax to bypass this.

The objection confuses "can create a registry" with "can guarantee all items are in the registry." Static languages enable the former; Python enforces the latter. This is the difference between DOF ≥ 2 and DOF $= 1$.

Concern 11: This Is Just Subtyping. *Potential concern:* The axis-parametric proposal ($T(\text{scope} = X)$ vs $T(\text{scope} = Y)$) is really about subtyping, and the author is confused about basic type theory.

Analysis: This objection conflates two orthogonal concerns:

1. **Type construction** (our claim): How is a type's *identity* determined at creation time?
2. **Subtyping** (not our claim): Given two types, can one substitute for the other?

Subtyping answers: "Can I use type A where type B is expected?" This presupposes types A and B already exist. It concerns *relationships between types*: $A <: B$, covariance, contravariance, Liskov substitution.

Axis-parametric construction answers: "What *is* this type?" It concerns type *identity* at creation time. Our proposal extends `type(name, bases, namespace)` to `type(name, bases, namespace, **axes)`, where axes become part of the type's identity. $T(\text{scope} = X)$ and $T(\text{scope} = Y)$ are simply *different types*—not subtypes, not supertypes, just distinct.

The category error: Responding to "type constructors should accept semantic axes" with "you're confused about subtyping" is like responding to "cars should have different engine options" with "you're confused about traffic laws." Construction and relationship are different layers.

Where subtyping *could* enter: One might later ask: "Is $T(\text{scope} = X) <: T(\text{scope} = Y)$?" This is a valid question—but it comes *after* our proposal, not instead of it. We make no claims about subtype relationships between axis-instantiated types. The axes determine identity; subtyping relationships are a separate design choice.

The pattern-match failure: The objection likely arises from pattern-matching "type parameters" $\to$ "must be about variance/subtyping." But type parameters in generics (e.g., `List[T]`) are indeed subject to variance analysis. Semantic axes are not type parameters in this sense—they are construction-time metadata that become part of type identity, analogous to how `bases` in `type(name, bases, ns)` determines identity without being "about subtyping."

A.3 Formal Verification Status

All core theorems are machine-checked in Lean 4:

- 6300+ lines of Lean code
- 190+ theorems verified
- 0 `sorry` placeholders
- 0 axioms beyond standard Lean foundations (`propext`, `Quot.sound`, `Classical.choice` only)

The Lean formalization is publicly available for verification.

B Historical and Methodological Context

B.1 On the Treatment of Defaults

Duck typing was accepted as "Pythonic" without formal justification. This asymmetry (conventions often require no proof, while changing conventions demands proof) is a methodological observation about community standards, not a logical requirement. The theorems in this paper provide the formal foundation that was absent from the original adoption of duck typing as a default.

B.2 Why Formal Treatment Was Delayed

Prior work established qualitative foundations (Malayeri & Aldrich 2008, 2009; Abdelgawad & Cartwright 2014; Abdelgawad 2016). We provide the first machine-verified formal treatment of typing discipline selection.

References

[1] Martín Abadi and Luca Cardelli. *A theory of objects*. Springer, 1996.

[2] Walid Taha Abdelgawad. Noop: A domain-theoretic model for object-oriented programming. *Electronic Notes in Theoretical Computer Science*, 301:3–21, 2014.

[3] Walid Taha Abdelgawad. Why nominal-typing matters in oop. arXiv preprint arXiv:1602.01047, 2016.

[4] Anonymous. Formal foundations for the single source of truth principle. Zenodo, 2025. Paper 2 of this series. Proves coherence impossibility theorems for programming languages.

[5] Kim Barrett, Bob Cassels, Paul Haahr, David A Moon, Keith Playford, and P Tucker Withington. A monotonic superclass linearization for dylan. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 69–82, 1996.

[6] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *ECOOP*, 2014.

[7] Manuel Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the ACM (JACM)*, 14(2):322–336, 1967.

[8] Manuel Blum. On the size of machines. *Information and Control*, 11(3), 1967.

[9] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981. Provides data on cost of error fixing vs development phase – earlier detection is cheaper (10x-100x cost multiplier for late vs early detection).

[10] Luca Cardelli. A semantics of multiple inheritance. *Information and computation*, 76(2-3):138–164, 1988.

[11] Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested refinements for dynamic languages. arXiv preprint, 2011. arXiv:1103.5055.

[12] Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested refinements: A logic for duck typing. In *POPL*, 2012.

[13] William R Cook, Walter Hill, and Peter S Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135, 1990.

[14] Robertas Damaševičius and Vytautas Štuikys. Assessment of the relative complexity and difficulty of software development methods. *Information Technology and Control*, 2010.

[15] Robertas Damaševičius and Vytautas Štuikys. Complexity metrics for metaprograms. *Computer Science and Information Systems*, 7(4):769–787, 2010.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[17] Joseph Gil and Itay Maman. Whiteoak: Introducing structural typing into java. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 73–90, 2008.

[18] INRIA / OCaml documentation contributors. The object layer (ocaml objects; structural object types and row variables). OCaml documentation, n.d. Online: caml.inria.fr.

[19] Ivan Ivannikov et al. PEP 544 – protocols: Structural subtyping (static duck typing). Python Enhancement Proposals, 2017. Online: peps.python.org/pep-0544/.

[20] Antoine Lamaison. *Inferring Useful Static Types for Duck Typed Languages*. PhD thesis, Imperial College London, 2012.

[21] Barbara H Liskov and Jeannette M Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[22] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *European Conference on Object-Oriented Programming*, pages 260–284. Springer, 2008.

[23] Donna Malayeri and Jonathan Aldrich. Cz: multiple inheritance without diamonds. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 21–40, 2009.

[24] Donna Malayeri and Jonathan Aldrich. Is structural subtyping useful? an empirical study. In *Programming Languages and Systems: 18th European Symposium on Programming, ESOP 2009*, pages 95–111. Springer, 2009.

[25] MDN contributors. Inheritance and the prototype chain. MDN Web Docs, n.d. Online: developer.mozilla.org.

[26] microsoft/TypeScript contributors. Nominal typing / nominal types request (issue #202). GitHub issue, 2014. Online: github.com/microsoft/TypeScript/issues/202.

[27] microsoft/TypeScript contributors. Proposal/experiment for unique/nominal-ish types (pr #33038). GitHub pull request, 2019. Online: github.com/microsoft/TypeScript/pull/33038.

[28] NASA IV&V Facility. Error cost escalation through the project life cycle. Technical report, NASA, 2010. Updated figures on relative error correction costs: bugs found at requirements/design time cost 1x to fix; at testing 10x; post-release 50x-100x.

[29] Oracle. The java language specification, §4.6 type erasure. Java SE Specification, 2014. Online: docs.oracle.com/javase/specs/jls/.

[30] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[31] Python Software Foundation. Built-in functions: `hasattr`. Python documentation, n.d. Online: docs.python.org.

[32] John C Reynolds. Types, abstraction and parametric polymorphism. In *Information processing*, volume 83, pages 513–523, 1983.

[33] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. *Scheme and Functional Programming Workshop*, 6:81–92, 2006.

[34] Tristan Simas. Lean 4 formalization: Typing discipline selection theorems. Supplementary material, 2025. 6300+ lines, 190+ theorems, 0 `sorry` placeholders. Included with paper submission.

[35] Tristan Simas. OpenHCS: Open-source high-content screening platform. Software, GitHub repository, 2025. Version 0.1.0. Available at: https://github.com/trissim/openhcs.

[36] Tristan Simas. UI Anti-Duck-Typing Refactor: ABC-based architecture (PR #44). GitHub pull request, 2025. Merged November 29, 2025. Available at: https://github.com/trissim/openhcs/pull/44.

[37] The Go Authors. The go programming language specification. Language specification, n.d. Online: go.dev/ref/spec.

[38] TypeScript Team. Type compatibility. TypeScript Handbook, n.d. Online: typescriptlang.org/docs/handbook/type-compatibility.html.

[39] Guido van Rossum et al. PEP 484 – type hints. Python Enhancement Proposals, 2014. Online: peps.python.org/pep-0484/.

[40] Todd L Veldhuizen. Tradeoffs in metaprogramming. *ACM SIGPLAN Notices*, 41(7):34–40, 2006.

[41] Philip Wadler. Linear types can change the world! In *Programming concepts and methods*, volume 3, page 5. Citeseer, 1990.