Mechanizing CatDB: A Lean Kernel for Typed Paths and Equation-Preserving Schema Mappings

Matthew Long
The YonedaAI Collaboration
YonedaAI Research Collective
Chicago, IL
matthew@yonedaai.com · https://yonedaai.com

23 July 2026

Abstract

A formalization can make a database claim more precise, but it can also make a small result sound larger than it is. We report the first Lean slice of CatDB, a proposed semantic data platform in which schemas and mappings are first-class objects. The checked kernel contains a typed directed graph, endpoint-indexed paths, a supplied equivalence relation on parallel paths, raw schema mappings, and mappings carrying a proof that they preserve that relation. Lean 4.32.1 checks empty-path identities, typing and associativity of path composition, lawfulness of identity and composite mappings, and mapping unit and associativity laws under a stated extensional equality.

The result is deliberately narrow. The theorem called `comp_typed` is a reflexive witness whose real content lies in the dependent type of composition. A schema presentation supplies its path equivalence and its laws; the library does not generate or decide a congruence. The checked model has no finiteness witness, attributes, scalar domains, database instances, data migration, queries, SQL, backends, transactions, or distributed semantics. It also has no refinement proof connecting Lean objects to the Rust runtime, Haskell reference model, or serialized fixtures.

We give the exact source-level statements for the six program invariants `PATH-02`–`PATH-04` and `MAP-03`–`MAP-05`, audit their assumptions, and reproduce the build in an isolated directory. The build reports no axiom dependencies for nine covered declarations; three audit-specific Python tests and the full 19-test utility suite pass. These facts are MACHINE-CHECKED or EXECUTABLE AUDIT EVIDENCE only at the named boundary. The paper's contribution is a reproducible formalization boundary and a roadmap toward finite instances, pullback migration, cross-language refinement, and selected query rewrites, not a claim that the CatDB database system is verified.

Contents

1 Introduction

1.1 Plain-language problem

A database system can pass thousands of tests while still leaving its most important semantic assumptions implicit. Consider a mapping that sends a source relationship to a longer path in a target schema. The implementation must know that the path starts and ends at the right objects. It must preserve the source equations that express semantic agreement between paths. When two mappings are composed, the result should preserve those equations as well. These are small laws, but they sit underneath migration, query rewriting, and integration.

Proof assistants offer one way to make such laws explicit. Yet the phrase "formally verified database" is dangerous when the checked artifact contains only a few mathematical definitions. A proof about path append does not verify SQL generation. A proof that two abstract mapping constructions are extensionally equal does not verify their serialized identifiers, provenance, or physical effects. A theorem accepted by Lean says exactly what its type says and nothing about unmodeled code.

This paper therefore treats formalization scope as part of the result. We describe the smallest accepted Lean kernel in the CatDB research program, quote its checked statements, expose its assumptions, reproduce its axiom inventory, and draw a hard line around everything that remains outside the model.

1.2 Research question

The research question is:

> *What is the smallest* CatDB *semantic kernel that can be represented and checked in Lean without implying that the database engine, its serialized artifacts, or its physical plans have been verified?*

The answer in this slice is: typed paths and equation-preserving schema mappings. More precisely, the library checks path identities, endpoint alignment through indices, path associativity, mapping preservation by identity and composition, and mapping category laws under a chosen extensional relation. It does not yet check data.

1.3 Contributions

This paper makes five contributions.

1. It gives a source-faithful account of a 346-line Lean kernel for typed paths and mappings. The account distinguishes definitions, proof fields, theorems, and build evidence.
2. It maps six CatDB program invariants to nine declarations and states every assumption needed for their interpretation.
3. It reproduces the formal build under the pinned Lean 4.32.1 toolchain, records the transitive axiom inventory, and separates kernel evidence from a Python source-hygiene audit.
4. It compares the kernel with general category theory, mathlib, and categorical database theory. The elementary mathematics is prior art; the contribution is the checked project boundary and its reproducibility discipline.
5. It gives a staged formalization roadmap whose next proof obligations are finite instances, pullback migration, cross-language refinement, and a deliberately small query-rewrite fragment.

1.4 Evidence labels

We use evidence labels literally.

- MACHINE-CHECKED means that the exact Lean declaration was accepted by the pinned toolchain and is named in the coverage inventory.
- EXECUTABLE AUDIT EVIDENCE means that a command or test ran successfully. It does not turn the tested program into a theorem.
- SUPPORTED BY PRIOR LITERATURE means that the underlying idea is established in the cited primary literature.
- ENGINEERING HYPOTHESIS marks a proposed implementation or formalization step.
- OPEN CONJECTURE marks a proposition for which this work supplies no proof.
- NOT CLAIMED marks an attractive inference that the paper explicitly rejects.

No theorem in this paper is labeled merely "proved" when its authority is a Lean declaration. The machine-checked label points to the checked statement, not to a prose paraphrase.

1.5 Result boundary

Figure 1 summarizes the boundary. The top box is the entire machine-checked semantic object. The lower boxes are planned program layers, not consequences of the top box.

2 A concrete example

Suppose a source schema has objects Order, $\mathsf{Customer}$, and Region, with generators

$$\mathsf{buyer} : \mathsf{Order} \to \mathsf{Customer}, \qquad \mathsf{region} : \mathsf{Customer} \to \mathsf{Region}.$$

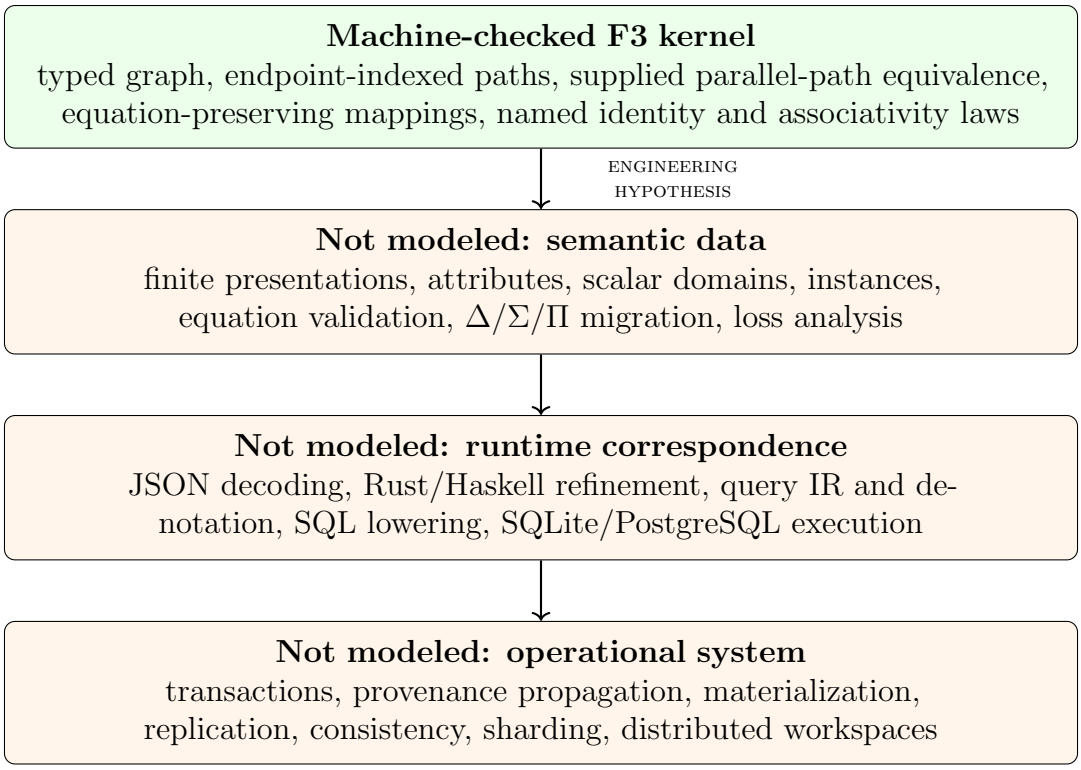

Figure 1: The checked kernel is a prerequisite for later layers, not a verification of them.

The path

$$\mathsf{buyer};\mathsf{region} : \mathsf{Order} \to \mathsf{Region}$$

is well typed because the target of buyer matches the source of region. A path that attempts to traverse region first from Order cannot be constructed in the intrinsic Lean representation.

Now let a target schema represent the same meaning with different objects and longer routes:

$$\mathsf{Purchase} \xrightarrow{\mathsf{account}} \mathsf{Account} \xrightarrow{\mathsf{owner}} \mathsf{Party} \xrightarrow{\mathsf{territory}} \mathsf{Territory}.$$

A raw mapping can send Order to Purchase, Customer to Party, and the generator buyer to the two-step target path account; owner. The path indices ensure that the image starts at Purchase and ends at Party.

If the source declares two parallel paths equivalent, lawfulness asks for evidence that their mapped target paths are equivalent too. The current Lean slice represents that evidence as a field of the mapping. It does not search for a proof, inspect a database, or decide a presented equational theory.

This example illustrates both the value and the limit of the encoding. A misaligned path is excluded from the intrinsic model. A law-preserving mapping must carry a proof. But a JSON document with misspelled identifiers can still exist outside Lean, and no theorem here says that a production decoder rejects it. Likewise, no rows move from a source table to a target table in this example. That operational work belongs to later layers.

3 Formal kernel

3.1 Typed directed graphs

The kernel begins with a universe-polymorphic typed multigraph.

Listing 1: The checked graph structure.

```
universe u

structure Graph where
  Obj : Type u
  Hom : Obj -> Obj -> Type u
```

For each pair of objects A, B, the type $G.\mathrm{Hom}\, A\, B$ contains the generating arrows from A to B. Parallel generators need not be equal. The definition requires neither finite objects nor finite hom-types. It also does not require decidable equality.

Remark 3.1 (Universe choice). Objects and generating arrows inhabit the same universe level in this slice. Nothing in the covered theorems needs finiteness, enumeration, or `DecidableEq`. This simplicity is an explicit modeling choice, not a database schema implementation.

3.2 Endpoint-indexed paths

Paths form an indexed inductive family.

Listing 2: Endpoint-indexed paths.

```
inductive Path (G : Graph) : G.Obj -> G.Obj -> Type u
  | nil (A : G.Obj) : Path G A A
  | cons {A B C : G.Obj} :
      G.Hom A B -> Path G B C -> Path G A C
```

The indices A and B are part of the type $\mathrm{Path}_G(A, B)$. The `nil` constructor produces the empty path at A. The `cons` constructor prepends a generator $A \to B$ to a path $B \to C$. This is a standard use of indexed inductive families in dependent type theory [9, 1].

Composition follows traversal order:

Listing 3: Path composition.

```
def comp {G : Graph} {A B C : G.Obj} :
    Path G A B -> Path G B C -> Path G A C
  | .nil _, q => q
  | .cons f rest, q => .cons f (comp rest q)
```

Thus `comp p q` means first traverse p, then q. The shared middle object B occurs in both input types. A caller inside this model cannot supply an $A \to B$ path and a $C \to D$ path unless Lean can identify B and C.

3.3 Schema presentations

The schema structure adds a relation between parallel paths.

Listing 4: The checked schema-presentation boundary.

```
structure SchemaPresentation where
  graph : Graph
  Equivalent :
    {A B : graph.Obj} ->
      Path graph A B -> Path graph A B -> Prop
  equivalent_refl :
    forall {A B} (p : Path graph A B), Equivalent p p
  equivalent_symm :
    forall {A B} {p q : Path graph A B},
      Equivalent p q -> Equivalent q p
  equivalent_trans :
    forall {A B} {p q r : Path graph A B},
      Equivalent p q -> Equivalent q r -> Equivalent p r
  equivalent_comp :
    forall {A B C} {p q : Path graph A B}
      {r s : Path graph B C},
      Equivalent p q -> Equivalent r s ->
        Equivalent (Path.comp p r) (Path.comp q s)
```

This definition deserves careful interpretation. It packages:

1. a typed graph;
2. a chosen relation on every family of parallel paths;
3. witnesses that the relation is reflexive, symmetric, and transitive;
4. a witness that it is compatible with composition.

It does not package a finite equation list. It does not construct the least congruence containing such a list. It does not offer a decision procedure for `Equivalent`. The relation and its laws are supplied by a value of the structure.

Category theory ordinarily treats associativity and identity laws as part of a category and treats functors as preserving identities and composition [7, 11]. The present representation is closer to a typed path syntax together with a compatible equivalence relation. A quotient category is mathematically nearby, but it is not constructed in the source.

3.4 Raw mappings

A raw mapping acts on objects and generating arrows:

Listing 5: Raw mappings.

```
structure RawMapping (S : Graph) (T : Graph) where
  onObj : S.Obj -> T.Obj
  onHom :
    {A B : S.Obj} -> S.Hom A B ->
      Path T (onObj A) (onObj B)
```

The image of a source generator may be a composite target path. The path indices encode its required endpoints. The recursive function `mapPath` extends the generator action to every source path:

Listing 6: Extension from generators to paths.

```
def mapPath (F : RawMapping S T) :
    Path S A B -> Path T (F.onObj A) (F.onObj B)
  | .nil _ => .nil (F.onObj A)
  | .cons f rest =>
      Path.comp (F.onHom f) (mapPath F rest)
```

Three supporting lemmas establish that mapping respects path composition, that the identity raw mapping acts identically on paths, and that path mapping through a composite raw mapping equals sequential path mapping.

3.5 Extensional equality

The mapping category laws do not use Lean's structural equality on `RawMapping`. They use:

Listing 7: Extensional equality of raw mappings.

```
def ExtensionalEq (F G : RawMapping S T) : Prop :=
  F.onObj = G.onObj /\
    forall {A B} (f : S.Hom A B),
      HEq (F.onHom f) (G.onHom f)
```

Equality of object actions is paired with heterogeneous equality of generator images. Heterogeneous equality is appropriate because those images inhabit path types whose indices depend on the object action.

Remark 3.2 (What has not been bundled). The source does not prove that `ExtensionalEq` is reflexive, symmetric, and transitive. It does not prove that composition respects it in both arguments. It therefore does not yet build a mathlib `Category` whose morphisms are equivalence classes of lawful mappings. The MAP-05 results are the three named propositions only.

3.6 Lawful mappings

A lawful mapping extends a raw mapping with a preservation witness:

Listing 8: Equation-preserving mappings.

```
structure LawfulMapping
    (S : SchemaPresentation)
    (T : SchemaPresentation)
    extends RawMapping S.graph T.graph where
  preserves :
    forall {A B} {p q : Path S.graph A B},
      S.Equivalent p q ->
        T.Equivalent
          (toRawMapping.mapPath p)
          (toRawMapping.mapPath q)
```

The definition expresses a semantic obligation exactly: source-equivalent paths map to target-equivalent paths. It does not decide that obligation. Constructing a lawful mapping requires the caller to provide a term of the `preserves` field.

4 Machine-checked path coverage

4.1 PATH-02: identities

The first invariant is represented by two theorems.

```
theorem id_comp (p : Path G A B) :
    comp (id G A) p = p := by
  rfl

theorem comp_id (p : Path G A B) :
    comp p (id G B) = p := by
  induction p with
  | nil _ => rfl
  | cons f rest ih =>
      simp only [comp]
      rw [ih]
```

Status. MACHINE-CHECKED.

The left identity follows by reduction of `comp`. The right identity is proved by induction because `comp` recurses on its first argument. The result holds for every value of the abstract `Graph` structure.

The original program-language statement referred to an object belonging to a schema and an empty serialized path. The Lean result does not model membership in a finite schema artifact. Its exact content is the algebraic identity of the indexed path constructors.

4.2 PATH-03: typed composition

The covered theorem is:

```
theorem comp_typed
    (p : Path G A B) (q : Path G B C) :
    Exists (fun r : Path G A C => r = comp p q) :=
  Exists.intro (comp p q) rfl
```

Status. MACHINE-CHECKED, with the qualification below.

The proof supplies `comp p q` and reflexivity. As a proposition, the theorem is immediate. The substantive typing guarantee appears in the type of `comp`:

$$\mathrm{Path}_G(A, B) \to \mathrm{Path}_G(B, C) \to \mathrm{Path}_G(A, C).$$

The library's coverage record says that endpoint alignment is represented in types. It does not say that the theorem decides whether two serialized identifiers match.

Remark 4.1 (Why the qualification matters). Calling `comp_typed` a nontrivial typing-soundness theorem would be misleading. There is no extrinsic syntax, typing judgment, or failing input in the model. Ill-aligned inputs cannot be given to `comp`; the result type already carries the outer endpoints. The theorem simply reifies that function application as an existential witness.

4.3 PATH-04: associativity

The associativity theorem is:

```
theorem comp_assoc
    (p : Path G A B)
    (q : Path G B C)
    (r : Path G C D) :
    comp (comp p q) r =
      comp p (comp q r) := by
  induction p with
  | nil _ => rfl
  | cons f rest ih =>
      simp only [comp]
      rw [ih]
```

Status. MACHINE-CHECKED.

The induction again follows the recursive structure of the first path. This is the ordinary associativity of list-like append, indexed by endpoints. It does not establish that reassociating a query plan preserves bag multiplicity, null behavior, errors, ordering, cost, or backend execution. Those properties require a query denotation that the library does not contain.

5 Machine-checked mapping coverage

5.1 MAP-03: lawful identity

For every schema presentation S, the source constructs:

```
def LawfulMapping.id
    (S : SchemaPresentation) :
    LawfulMapping S S where
  toRawMapping := RawMapping.id S.graph
  preserves := by
    intro A B p q h
    change S.Equivalent
      (RawMapping.mapPath
        (RawMapping.id S.graph) p)
      (RawMapping.mapPath
        (RawMapping.id S.graph) q)
    rw [RawMapping.mapPath_id,
        RawMapping.mapPath_id]
    exact h
```

Status. MACHINE-CHECKED.

The proof starts with a supplied hypothesis $h : S.\mathrm{Equivalent}\, p\, q$. After rewriting identity path mapping, the goal is exactly h. No equational decision procedure is invoked.

The result excludes serialization details. A production identity mapping may have an artifact identifier, schema-version fields, provenance, or display metadata. None of those fields appears in the Lean object, so none is covered.

5.2 MAP-04: lawfulness under composition

For lawful mappings $F : S \to T$ and $G : T \to U$, the checked constructor is:

```
def LawfulMapping.comp
    (F : LawfulMapping S T)
    (G : LawfulMapping T U) :
    LawfulMapping S U where
  toRawMapping :=
    RawMapping.comp F.toRawMapping G.toRawMapping
  preserves := by
    intro A B p q h
    have mappedByF := F.preserves h
    have mappedByG := G.preserves mappedByF
    change U.Equivalent
      (RawMapping.mapPath
        (RawMapping.comp
          F.toRawMapping G.toRawMapping) p)
      (RawMapping.mapPath
        (RawMapping.comp
          F.toRawMapping G.toRawMapping) q)
    rw [RawMapping.mapPath_compMapping,
        RawMapping.mapPath_compMapping]
    exact mappedByG
```

Status. MACHINE-CHECKED.

The proof is compositional in a precise but conditional sense. It assumes that F and G already carry preservation proofs. It first transports the source relation through F, then transports the resulting target relation through G, and finally rewrites the raw path action of the composite.

This is not an algorithm for discovering a schema mapping. It is not an algorithm for proving a candidate mapping lawful. It does not inspect source records, move data, or detect information loss. It shows closure of the proof-carrying construction.

5.3 MAP-05: units and associativity

The lawful-mapping namespace exposes three theorems:

```
theorem LawfulMapping.comp_id_ext
    (F : LawfulMapping S T) :
  RawMapping.ExtensionalEq
    (comp F (id T)).toRawMapping
    F.toRawMapping

theorem LawfulMapping.id_comp_ext
    (F : LawfulMapping S T) :
  RawMapping.ExtensionalEq
    (comp (id S) F).toRawMapping
    F.toRawMapping

theorem LawfulMapping.comp_assoc_ext
    (F : LawfulMapping S T)
    (G : LawfulMapping T U)
```

```
  (H : LawfulMapping U V) :
RawMapping.ExtensionalEq
  (comp (comp F G) H).toRawMapping
  (comp F (comp G H)).toRawMapping
```

Status. MACHINE-CHECKED.

The proofs delegate to the corresponding raw-mapping results. Object actions are definitionally equal. Generator images are shown heterogeneously equal using the support lemmas for mapping paths through identities and composites.

Invariant	Checked proposition	Excluded interpretation
`MAP-03`	Identity raw action plus a preservation proof forms a lawful mapping.	Canonical serialized identity artifact or attribute/value mapping.
`MAP-04`	Two proof-carrying lawful mappings compose to another.	Automatic proof search, data migration, or information-loss analysis.
`MAP-05`	Unit and associativity propositions under the exact `ExtensionalEq`.	Equality of mapping records, artifact IDs, provenance, or a fully bundled category.

Table 1: Mapping coverage and its nearest tempting overclaim.

6 What the schema assumptions mean

6.1 The equivalence relation is an input

The field S.Equivalent can be read as the semantic equality used by the formal profile. Its equivalence and composition-congruence properties are structure fields. Every `SchemaPresentation` value must supply them. The mapping theorems are parametric in that supplied relation.

This design is enough to state preservation cleanly. It avoids committing the first slice to a particular equational prover. It also means that the current formalization does not answer a central executable question: given a finite list of path equations, how should the runtime establish equality of two paths?

One future answer may construct the generated congruence inductively. Another may define a normalization algorithm for a restricted terminating rewrite system. A third may use proof-carrying mapping artifacts. Each choice has different completeness, decidability, and performance properties. The present code deliberately proves none of them.

6.2 No finiteness or decidable equality

Database schemas are finite artifacts in the implementation roadmap, but `Graph.Obj` and `Graph.Hom` are arbitrary types. The theorems are therefore more general in one dimension

and less executable in another. They apply without a finiteness assumption, but they do not produce enumeration, validation, or diagnostics.

This difference matters for interpretation. A production validator may need to check that every arrow endpoint resolves and that identifiers are unique. Those are representation-level obligations. The indexed Lean model begins after valid objects and arrows already exist.

6.3 No attributes or values

The graph has only objects and arrows. It does not distinguish entity types from value types, nor does it model attributes, integer ranges, decimal lexemes, UUIDs, nullability, cardinality, or identity rules. Algebraic database models explicitly address typeside operations and concrete data [12]. This Lean slice does not.

6.4 No instances

Categorical database theory commonly models an instance as a set-valued functor from a schema category [13, 14]. There is no `Instance` declaration in the audited source. Consequently:

- path equality has no pointwise data interpretation;
- the library cannot state that a database satisfies a schema equation;
- no migration functor acts on data;
- no query returns records;
- no finite validator is sound or complete.

These are roadmap items, not hidden corollaries.

7 Build, axiom inventory, and trust

7.1 Pinned environment

The project pins:

Listing 9: `lean-toolchain`.

```
leanprover/lean4:v4.32.1
```

The Lake package sets `autoImplicit` to false and warnings as errors. Its manifest has no external packages. The official Lake documentation describes `lean-toolchain`, the package configuration, the manifest, and `.lake` build artifacts [10]. The absence of external packages keeps this slice small; it does not make Lean's standard library or toolchain disappear from the trusted base.

7.2 Isolated reproduction

For this paper, the formal sources and configuration were copied to a newly created temporary directory. The build did not reuse the repository's `.lake` directory. The observed compiler was:

```
Lean (version 4.32.1, arm64-apple-darwin24.6.0,
commit f054605aea4b840552cca2e725580bffd1e1b704, Release)
```

The clean build completed seven jobs successfully.

7.3 Transitive axiom inventory

`CatDB.Coverage` executes `#print axioms` for nine declarations. Lean's reference manual states that this command reports the axioms used by a declaration transitively [8]. The isolated build reported:

```
'CatDB.Path.id_comp' does not depend on any axioms
'CatDB.Path.comp_id' does not depend on any axioms
'CatDB.Path.comp_typed' does not depend on any axioms
'CatDB.Path.comp_assoc' does not depend on any axioms
'CatDB.LawfulMapping.id' does not depend on any axioms
'CatDB.LawfulMapping.comp' does not depend on any axioms
'CatDB.LawfulMapping.comp_id_ext' does not depend on any axioms
'CatDB.LawfulMapping.id_comp_ext' does not depend on any axioms
'CatDB.LawfulMapping.comp_assoc_ext' does not depend on any axioms
```

Status. MACHINE-CHECKED build evidence for these declarations and this source state.

An empty reported inventory rules out transitive use of user axioms and `sorryAx` in the covered declarations. It does not eliminate the trusted computing base. We trust the Lean 4.32.1 elaborator and kernel, the standard library needed by these declarations, the platform on which they ran, and the accuracy with which this paper maps prose to source. Lean 4's system design is described by de Moura and Ullrich [6]; dependent type checking remains the formal authority, not the manuscript.

7.4 Why source scans are secondary

The checked-in Python audit strips comments and strings, rejects `sorry`, `admit`, `sorryAx`, and declarations introduced by `axiom`, then searches for required symbol substrings. Three unit tests cover placeholder rejection, comment/string handling, and unterminated block comments.

The audit is useful because it fails quickly on common proof holes. It is not a Lean parser. It does not compare fully elaborated theorem types with a coverage contract. Required-name substring checks can be spoofed. Its prohibited-token list is narrower than a comprehensive trusted-code policy. For these reasons the kernel's axiom inventory is normative, while the Python script is source hygiene.

Evidence	What it establishes	What it does not establish
Lean elaboration and kernel checking	The declaration has the checked type under the toolchain.	The prose means the same thing, or production code refines the definition.
`#print axioms`	Transitive axiom dependencies reported by Lean.	Absence of bugs in Lean or correctness of unmodeled components.
Python source audit	Named tokens are absent after its lexical stripping, and names appear.	Theorem semantics, axiom freedom, or a secure trusted-code policy.
Python unit tests	The audit behaves as expected on three cases.	Coverage of all lexical edge cases or Lean constructs.
Historical code review	An independent reviewer reproduced and critiqued the slice.	Peer review of this paper or permanent correctness of later source.

Table 2: Evidence sources are complementary, not interchangeable.

8 Relationship to prior work

8.1 Lean and mathlib

Lean is both an interactive theorem prover and a functional programming language based on dependent type theory [5, 6]. Indexed inductive families are a standard Lean mechanism; the endpoint-indexed `Path` is a direct use of that mechanism [9].

The mathlib library contains general category, functor, and natural transformation infrastructure [15, 16]. The current CatDB package has no external Lean dependency, including no mathlib dependency. It uses a small bespoke structure because the F3 gate targets specific path and mapping invariants. This choice reduces the dependency surface, but it leaves general categorical bundling and interoperability for later work.

8.2 Category theory

Identity, associativity, functors, and composition are elementary category theory [7, 11]. The checked path theorems resemble the laws of a free category on a graph. The supplied equivalence relation resembles the congruence needed to pass from path syntax to a presented category. However, the source neither constructs the free category as a library object nor forms a quotient by a generated congruence.

Accordingly, the paper claims no new mathematical theorem. Its value is engineering precision: the CatDB invariant IDs are tied to terms that a kernel checks, and the assumptions are visible.

8.3 Categorical databases

Schemas as categories, instances as functors, and data migration along schema mappings are established ideas. Spivak develops functorial data migration [13]. Spivak and Wisnesky connect the categorical construction to relational foundations and describe the induced Σ_F, Δ_F, and Π_F functors [14]. Algebraic databases extend the set-valued-functor model with typeside algebra for concrete data [12]. Categorical data integration has also been demonstrated in scientific applications [2].

The current Lean library stops before all of those instance-level constructions. It mechanizes only typed paths and preservation structure. In particular, no result about Δ migration should be attributed to this paper even though such a result is a planned next step.

8.4 Mechanized query reasoning

Cosette and HoTTSQL show that meaningful query-equivalence verification requires a query language and a carefully defined semantics [3, 4]. SQL multiplicities, nulls, correlated subqueries, aggregation, and other features cannot be reduced to path associativity without additional modeling.

The contrast is useful. `PATH-04` may eventually support a proof about a typed path-navigation rewrite. It does not currently express a SQL query, much less prove an optimizer correct.

8.5 Novelty assessment

Table 3: Novelty ledger.

Item	Status	Assessment
Endpoint-indexed paths	SUPPORTED BY PRIOR LITERATURE	Standard indexed-family technique applied to the CatDB kernel.
Path identity and associativity	SUPPORTED BY PRIOR LITERATURE and MACHINE-CHECKED	Elementary mathematics, checked in this exact encoding.
Equation-preserving schema mappings	SUPPORTED BY PRIOR LITERATURE and MACHINE-CHECKED	Categorical preservation is established; the project-specific structure and proof terms are checked.

Item	Status	Assessment
Mapping unit and associativity laws	SUPPORTED BY PRIOR LITERATURE and MACHINE-CHECKED	Checked under the exact `ExtensionalEq`, without a bundled category.
Axiom-visible invariant coverage	EXECUTABLE AUDIT EVIDENCE	Project-specific reproducibility practice that ties program IDs to build output.
Verified finite CatDB instances	NOT CLAIMED	No instance type exists in the audited source.
Verified data migration or query compilation	NOT CLAIMED	No migration, query, or backend semantics exists in the audited source.
Cross-language semantic equivalence	OPEN CONJECTURE	No refinement relation connects Lean with Rust, Haskell, or JSON.

9 Exact nonclaims

The following exclusions are part of the theorem statement at program scale.

9.1 Representation and validation

The library does not verify that:

- serialized schema identifiers are unique;
- arrow endpoints resolve;
- a JSON path is well typed;
- normalization is deterministic;
- diagnostic codes or JSON pointers are correct;
- artifact identifiers survive mapping composition.

These obligations concern a decoder and normalized runtime representation that are absent from the formal model.

9.2 Database semantics

The library has no:

- finite carriers or database records;
- total arrow interpretations;
- attributes or scalar values;

- pointwise equation-satisfaction relation;
- instance homomorphisms;
- bag, set, null, or error semantics.

Therefore it proves no fact about a database state.

9.3 Migration and integration

The library does not define Δ, Σ, or Π. It does not prove migration totality, identity, composition, adjunctions, or information-loss properties. It does not say that semantic mappings replace physical ETL, nor that a mapping can be written back safely.

9.4 Compilation and execution

The library does not define queries, logical plans, physical plans, SQL, SQLite, PostgreSQL, APIs, materialization, incremental maintenance, or cost models. A path theorem is not execution evidence.

9.5 Distributed semantics

The library does not define operations, replicas, clocks, causality, CRDTs, partitions, consensus, consistency levels, workspaces, or merge policies. Category theory does not remove distributed-systems tradeoffs, and this formalization does not model them.

9.6 Whole-system verification

The phrase "CatDB is verified" is false on the evidence in this paper. The defensible statement is:

> Lean 4.32.1 machine-checks the named typed-path and equation-preserving-mapping declarations at the recorded source hashes, and reports no transitive axiom dependencies for the nine covered declarations.

10 Formalization roadmap

10.1 R1: harden the current boundary

The first improvements should make the present claim easier to audit.

1. Replace required-name substring matching with a machine-readable coverage contract tied to elaborated declaration types.
2. Prove that `ExtensionalEq` is an equivalence relation.
3. Prove that raw mapping composition respects `ExtensionalEq`.

4. Bundle schemas and lawful mappings into a category only after the equality and congruence obligations are complete.
5. Add an independent `leanchecker` replay to the release gate.

Each item is an ENGINEERING HYPOTHESIS until implemented and checked.

10.2 R2: finite presentations and equation evidence

The current `SchemaPresentation` should not be overloaded with executable finiteness. A later layer should separately define:

- a finite collection of object and generator identifiers;
- a finite collection of parallel path equations;
- an intrinsic semantic graph obtained from validated input;
- the generated congruence or a named decidable fragment;
- soundness of any normalization or proof-search procedure.

Completeness should be claimed only for a precisely bounded fragment. General equality in finitely presented categories is not made decidable by calling the schema finite.

10.3 R3: attributes and finite instances

The next high-value semantic layer is a finite instance model. It should include:

1. one finite carrier for every entity;
2. total functions for every relationship generator;
3. a bounded scalar profile for attributes;
4. target-carrier membership and scalar typing;
5. pointwise satisfaction of path equations.

The principal theorem targets are soundness and completeness of the finite validator for its exact model. Completeness must not silently include nulls, partial arrows, SQL constraints, or infinite domains.

10.4 R4: pullback migration

Once instances exist, the first migration target is the contravariant pullback Δ_F. The intended obligations are:

- a lawful $F : S \to T$ and valid T-instance produce a valid S-instance;
- Δ_{id} is extensionally the identity;
- $\Delta_{G \circ F}$ agrees with $\Delta_F \circ \Delta_G$.

These are established categorical patterns [13, 14], but they remain OPEN CONJECTURE for the CatDB finite model.

Σ and Π should wait. Their practical formalization depends on quotients, representatives, finite computation, and complexity choices that the current program has not fixed.

10.5 R5: cross-language refinement

Shared fixture names are not a refinement proof. A credible correspondence among Lean, Haskell, Rust, and JSON requires:

1. versioned external syntax;
2. a decoder relation from syntax to semantic objects;
3. explicit error cases for malformed data;
4. agreement on composition order and equality;
5. separation of semantic values from artifact identifiers;
6. executable or proved correspondence claims.

The Haskell model can serve as an executable oracle and the Rust code as the production kernel, but neither role transfers Lean theorems automatically.

10.6 R6: selected query rewrites

Only after a query language and denotation are fixed should the project prove rewrites. A modest first target is reassociation of pure, total path navigation over finite instances. Filter fusion, projection pushdown, join reassociation, aggregation, and SQL lowering require explicit choices for nulls, errors, multiplicity, order, volatility, and backend capabilities.

Backend differential tests should remain separate from semantic proofs. Passing them is execution evidence; it is not a theorem about every backend version.

11 Falsification and adversarial checks

A formalization paper should say what would disprove its own claims.

11.1 Source-level falsifiers

The machine-checked coverage claim fails if:

- any quoted source statement differs from the elaborated declaration;
- the recorded source hashes do not reproduce the build;
- any covered declaration reports an axiom dependency;
- a later source change is discussed using stale build evidence;

- a theorem name remains while its type is weakened.

The final item is why name-substring checks cannot be the semantic gate.

11.2 Interpretive falsifiers

The paper's interpretation fails if it:

- calls `comp_typed` a runtime decision procedure;
- treats `Equivalent` as a generated or decidable congruence;
- calls `ExtensionalEq` categorical equality without proving its relation and congruence laws;
- treats proof-carrying lawfulness as automatic mapping validation;
- infers instance, migration, query, backend, or distributed behavior from the F3 kernel;
- says that a Python token scan proves theorems;
- says that exact mathematics implies cross-language implementation equivalence.

11.3 Counterexample-oriented extensions

Later slices should make negative evidence first class. An extrinsic path validator needs malformed endpoint cases. An instance validator needs single-fault inputs with missing assignments or broken equations. A query rewrite needs finite countermodel search alongside proof attempts. A cross-language refinement needs fixtures that distinguish composition order and semantic equality from artifact identity.

Formal proof is strongest when the modeled proposition is clear. Persisted counterexamples are strongest when the intended proposition was wrong or the implementation violated it. The research program needs both.

12 Reproducibility

12.1 Commands

The formal build can be reproduced from the repository root:

```
lake env lean --version
lake clean
lake build
python3 scripts/audit-lean.py lean
python3 -m unittest -v scripts.tests.test_audit_lean
PYTHONDONTWRITEBYTECODE=1 \
  python3 -m unittest discover \
  -s scripts/tests -p 'test_*.py' -v
```

For isolation, copy `lean-toolchain`, `lakefile.lean`, `lake-manifest.json`, the `lean/` directory, and the audit files to a new directory before running the first five commands. The manuscript build is separate and described in the accompanying `reproducibility.md`.

12.2 Expected outcomes

- Lean reports version 4.32.1.
- `lake build` completes seven jobs.
- Nine lines say that the named declarations do not depend on axioms.
- The Lean source audit reports coverage for `PATH-02 PATH-03 PATH-04 MAP-03 MAP-04 MAP-05`.
- Three audit-specific tests pass.
- Nineteen total Python utility tests pass.

Only the first five bullets concern the formal slice directly. Of the 19 Python tests, 16 test other fixture and paper-verification utilities.

12.3 Source digests

Table 4: Audited source digests.

Artifact	SHA-256
`lean-toolchain`	`8e3538e0ab5f81a3ee04927d8838c8c674e0e112838b4b3ce87ec218143276af`
`lakefile.lean`	`a1b6c42f458b66e446afba585cf8fcde183760945fe5092e8cb58ec96397a92b`
`lake-manifest.json`	`971337b5c015b73116da2cabcdc2b5356196d94066cb66417d62b9dc98aec523`
`lean/CatDB.lean`	`c809cc62214d231ccccf8ed40e3afc9e4959f6bdf9de2e0c17af56a2240ee417`
`lean/CatDB/Path.lean`	`9d6ee7ac1f6ad531dbe012834c2ff2a42f5cd59dfac9f6f875444578934f0a14`
`lean/CatDB/Schema.lean`	`20334a5e31668f056c3335e2cd8868cf965b172040e1f8ef21619de33bf04439`
`lean/CatDB/Mapping.lean`	`1b5de49323febb5448567bfb271ff86c98af6e90710658e19a35ea6308b15054`
`lean/CatDB/Coverage.lean`	`cecc2b89c5a7531c5cefbd946be3e630308f7edaadbde33d3a8c3f92d61ce773`
`lean/COVERAGE.md`	`35dd37698a6820cd166fa324edf70d3c682f9bc1d08e746317a6b1b26177f412`

Artifact	SHA-256
`scripts/audit-lean.py`	`dd682cafaf4fcafed967acc6a14e0fea0cdc2aaf8540358e86d8a8ff440890f7`
`scripts/tests/test_audit_lean.py`	`85d37aa83ae2a4ab25d880c43f33b16cc95a43685ba163dcdf0cfd526e56183f`

12.4 Reproducibility status

The fresh reproduction was performed on `arm64-apple-darwin24.6.0`. No external Lean packages are listed in the manifest. Reproduction on another supported platform should compare declaration types and axiom output, not binary build artifacts.

This manuscript itself remains unreviewed. Successful LaTeX compilation does not satisfy the program's peer-review gate and must not be treated as acceptance.

13 Limitations and open questions

13.1 A small theorem boundary

The accepted slice is intentionally small. It checks the algebra of a list-like indexed path syntax and the closure properties of proof-carrying mappings. That is useful infrastructure, but it is not yet the part of a database system that handles data.

13.2 Intrinsic versus extrinsic correctness

Intrinsic typing excludes malformed paths after they have entered the model. Production software still receives untrusted external syntax. The missing refinement layer must show that a decoder either produces an intrinsically well-typed value with the intended meaning or returns a precise error.

An intrinsic model can simplify later proofs. It cannot replace boundary validation.

13.3 Equality design

The supplied `Equivalent` relation avoids premature commitment, but it moves a hard problem into an assumption. The program must decide whether future mapping artifacts carry proofs, use a decidable equational fragment, or defer obligations for human review. Different schema classes may need different answers.

13.4 Extensional equality

`ExtensionalEq` is adequate for the three current MAP-05 statements, but it is not yet a finished quotient or setoid interface. Its interaction with proof fields, serialization identity, stable schema versions, and provenance is open.

13.5 No performance claim

The Lean package contains no benchmark. Proof checking time for this tiny kernel says nothing about schema-compilation latency, query-planning overhead, or physical execution. The claim that a categorical layer can remain outside the hot path is a systems hypothesis for other papers.

13.6 No cross-language theorem

Rust and Haskell components in the broader program may implement similarly named operations. The present paper has not proved that their definitions match Lean. Cross-checking shared fixtures can find disagreements, but only a stated refinement relation can say what agreement means.

13.7 Open questions

1. What finite presentation and congruence representation gives useful executable equality while preserving a clear completeness boundary?
2. Should lawful production mappings carry proof objects, validation certificates, or unresolved obligations?
3. What is the right extensional equality for mappings once version and provenance fields exist?
4. Can a finite-instance model support a simple, machine-checked Δ implementation that remains aligned with Rust and Haskell?
5. Which query rewrites have enough semantic value to justify Lean mechanization before a full relational semantics exists?
6. How should proof coverage be versioned when the executable schema profile changes?

14 Discussion

The main lesson is not that six invariants are difficult theorems. Most are elementary. The lesson is that mechanization forces a program to choose what its words mean.

"Typed path" can mean an external list that a validator checks, or an intrinsic value whose endpoints occur in its type. The current Lean slice chooses the latter. "Schema presentation" can mean a finite graph and equations, their generated category, or a graph equipped with a supplied equivalence relation. The current slice chooses the third. "Lawful mapping" can mean a record that passed an algorithm, or a mapping paired with a preservation proof. The current slice chooses proof-carrying lawfulness.

These choices are legitimate as long as their translation to the runtime is not assumed. The next research risk is not a failed induction proof. It is a semantic gap between intrinsic Lean objects and versioned production artifacts. That gap should be addressed with explicit decoding and refinement relations, not with stronger prose.

The result also clarifies the role of category theory in CatDB. Identity, composition, and preservation give a coherent algebra for schema mappings. They do not remove physical data movement, decide every mapping, or solve distributed consistency. Their value is compositional structure. Formal verification can protect that structure once the modeled boundary reaches the operations that matter.

15 Conclusion

The first CatDB Lean slice establishes a precise, reproducible base. Typed paths carry endpoints in their types. Empty paths are identities and composition is associative. Raw mappings extend from generators to paths. Lawful mappings carry equation-preservation evidence. Identity and composite mappings are lawful, and their unit and associativity propositions hold under the library's extensional equality.

That is the complete positive result. The isolated Lean 4.32.1 build reports no axiom dependencies for the nine covered declarations. The source audit and its tests pass. The theorem types, assumptions, commands, and hashes are recorded so that the claim can be checked again.

The larger CatDB thesis remains ahead. Finite schemas, attributes, instances, migration, query semantics, backend refinement, provenance, and distributed behavior are not verified here. The next milestone should be a finite-instance model and pullback migration, preceded by a stronger equality and coverage boundary. Until those artifacts exist, the honest conclusion is not that CatDB is verified. It is that a small semantic kernel has become exact enough to verify, criticize, and extend.

A Complete coverage inventory

Table 5: Exact program coverage.

ID	Declarations	Assumptions	Exact boundary
PATH-02	`Path.id_comp`, `Path.comp_id`	Arbitrary typed graph and a well-formed indexed path.	Left and right identity of the empty path.
PATH-03	`Path.comp_typed` and the type of `Path.comp`	Inputs share the same middle object in their indices.	An $A \to C$ result exists and equals composition; no external alignment decision.
PATH-04	`Path.comp_assoc`	Three endpoint-aligned indexed paths in an arbitrary graph.	Syntactic composition is associative.
MAP-03	`LawfulMapping.id`	The schema supplies a path equivalence with the stated laws.	Identity raw mapping preserves that relation.

ID	Declarations	Assumptions	Exact boundary
MAP-04	`LawfulMapping.comp`	Both mappings carry preservation witnesses.	The proof-carrying composite preserves the relation.
MAP-05	`LawfulMapping.comp_id_ext`, `LawfulMapping.id_comp_ext`, `LawfulMapping.comp_assoc_ext`	The exact object-action equality plus heterogeneous equality of generator images.	Two units and associativity under `ExtensionalEq`; no serialized IDs.

B Coverage source

The full coverage module is:

```
import CatDB.Mapping

#print axioms CatDB.Path.id_comp
#print axioms CatDB.Path.comp_id
#print axioms CatDB.Path.comp_typed
#print axioms CatDB.Path.comp_assoc
#print axioms CatDB.LawfulMapping.id
#print axioms CatDB.LawfulMapping.comp
#print axioms CatDB.LawfulMapping.comp_id_ext
#print axioms CatDB.LawfulMapping.id_comp_ext
#print axioms CatDB.LawfulMapping.comp_assoc_ext
```

The inventory includes the identity and composition constructors because they contain proof fields and are part of the machine-checked mapping coverage.

C Claim ledger

Table 6: Claim ledger.

Claim	Label	Evidence and qualification
Endpoint-indexed paths compose at a shared middle object.	MACHINE-CHECKED	Type of `Path.comp`; says nothing about JSON identifiers.
Empty paths are two-sided identities.	MACHINE-CHECKED	`id_comp`, `comp_id`; arbitrary graph.
Path composition is associative.	MACHINE-CHECKED	`comp_assoc`; syntactic paths only.
The PATH-03 theorem is nontrivial typing-soundness.	NOT CLAIMED	Its witness proof is immediate; the dependent function type is substantive.

Claim	Label	Evidence and qualification
Identity mappings preserve the supplied equivalence.	MACHINE-CHECKED	`LawfulMapping.id`; the relation and laws are structure fields.
Lawful mappings are closed under composition.	MACHINE-CHECKED	`LawfulMapping.comp`; both inputs already carry proofs.
Mapping units and associativity hold under the exact extensional relation.	MACHINE-CHECKED	Three MAP-05 theorems; no artifact fields.
`ExtensionalEq` is a proved composition congruence and equivalence relation.	NOT CLAIMED	Those theorems are absent.
The nine covered declarations are axiom-free in the audited build.	MACHINE-CHECKED	Transitive `#print axioms` reports at recorded hashes and toolchain.
The source audit proves theorem semantics.	NOT CLAIMED	It is lexical hygiene and name coverage only.
Finite database instances are verified.	NOT CLAIMED	No instance declaration exists.
Pullback migration is verified.	OPEN CONJECTURE	F4 roadmap target only.
Lean, Haskell, and Rust have equivalent semantics.	OPEN CONJECTURE	No refinement proof exists.
The database runtime is verified.	NOT CLAIMED	Runtime behavior is outside the formal model.

D Review checklist

Before this manuscript can advance from unreviewed status, reviewers should answer the following.

D.1 Formal methods review

1. Do all quoted Lean declaration types match the source?
2. Is every MACHINE-CHECKED statement limited to a checked declaration?
3. Is PATH-03 characterized without inflating its proof content?
4. Is `Equivalent` described as a supplied relation with laws?
5. Is `ExtensionalEq` described without assuming missing structure?
6. Is the axiom inventory interpreted according to Lean's documentation?

D.2 Category-theory review

1. Does the paper avoid claiming construction of a quotient category?
2. Are identity, associativity, functor-like mapping, and preservation placed correctly relative to prior literature?
3. Does the paper distinguish a practical schema presentation from an arbitrary graph with a supplied equivalence?

D.3 Database-systems review

1. Are external validation and intrinsic typing kept separate?
2. Are instances, migration, queries, and physical execution excluded?
3. Does the discussion avoid transferring formal laws to SQL, transactions, or distribution?
4. Is the proposed refinement roadmap testable?

D.4 Reproducibility review

1. Do the source hashes match?
2. Does the isolated build use Lean 4.32.1 and complete seven jobs?
3. Are nine empty axiom inventories printed?
4. Do the three audit-specific tests and the 19-test suite pass?
5. Does the manuscript compile without errors or material overfull boxes?
6. Has every PDF page been visually inspected?

References

[1] Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich. *Theorem Proving in Lean 4*. Lean Community, 2026.

[2] Kristopher Brown, David I. Spivak, and Ryan Wisnesky. Categorical data integration for computational science. *Computational Materials Science*, 164:127–132, 2019.

[3] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *8th Biennial Conference on Innovative Data Systems Research*, 2017.

[4] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. HoTTSQL: Proving query rewrites with univalent SQL semantics. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 510–524. ACM, 2017.

[5] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction – CADE-25*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015.

[6] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021.

[7] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer, 2 edition, 1998.

[8] Lean FRO. The Lean language reference: Axioms. https://lean-lang.org/doc/reference/latest/Axioms/, 2026. Accessed 2026-07-23.

[9] Lean FRO. The Lean language reference: Inductive types. https://lean-lang.org/doc/reference/latest/The-Type-System/Inductive-Types/, 2026. Accessed 2026-07-23.

[10] Lean FRO. The Lean language reference: Lake. https://lean-lang.org/doc/reference/latest/Build-Tools-and-Distribution/Lake/, 2026. Accessed 2026-07-23.

[11] Tom Leinster. *Basic Category Theory*. Cambridge University Press, 2014.

[12] Patrick Schultz, David I. Spivak, Christina Vasilakopoulou, and Ryan Wisnesky. Algebraic databases. *Theory and Applications of Categories*, 32(16):547–619, 2017.

[13] David I. Spivak. Functorial data migration. *Information and Computation*, 217:31–51, 2012.

[14] David I. Spivak and Ryan Wisnesky. Relational foundations for functorial data migration. *Journal of Functional Programming*, 25:e9, 2015.

[15] The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 367–381. ACM, 2020.

[16] The mathlib Community. Mathlib.CategoryTheory.Category.Basic. https://leanprover-community.github.io/mathlib4_docs/Mathlib/CategoryTheory/Category/Basic.html, 2026. Accessed 2026-07-23.