A Haskell Reference Semantics for CatDB Schema Mappings: Bounded Laws, Closed Fixtures, and a Rust Comparison

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

23 July 2026

Result status: ACCEPTED COMPUTATIONAL at source `abd655f`

Abstract

An executable reference model is useful only when its authority and limits are visible. This paper reports the accepted Slice 2 Haskell reference for CatDB, a research system for typed, versioned schema mappings. The program models a finite profile of schemas, typed paths, and total schema mappings. It validates names, endpoints, map totality, attribute types, and direct equation-preservation obligations; constructs identity mappings; and composes mappings by path substitution. Its fixture codec rejects unknown fields and round-trips every accepted JSON document exactly.

The result is tied to one reviewed source revision and one 24-case corpus. The Haskell program independently computed 16 semantic outcomes: three schema parsing, three path typing, two path composition, two identity mapping, and six mapping composition cases. Eight migration, information-loss, and schema-diff cases were decoded and round-tripped but explicitly reported as structural-only with no semantic output. Seven deterministic QuickCheck properties completed 1,550 successful trials over bounded linear-schema generators. A Rust process separately computed the same 16 outcomes, both implementations passed their own golden checks, and the comparator reported zero disagreements. An independent review of the exact source range found no blocker or major issue and issued a pass verdict. An isolated author rerun reproduced the accepted Haskell JSON byte-for-byte.

The evidence is computational, not deductive. The accepted equality procedure uses syntactic equality plus one directly declared equation; the property generators are narrow; and the accepted package had reproducibility and temporary-file hardening gaps. Later repository work adds finite instance validation and bounded pullback migration, but it does not retroactively widen this result. A still later working-tree candidate adds information-loss and schema-diff analysis, but it remains uncommitted and unreviewed.

This paper claims no full migration semantics, query semantics, backend execution, distributed behavior, performance result, general Rust/Haskell equivalence theorem, or proof of a categorical law.

Contents

1 Introduction

1.1 What an oracle must not conceal

Suppose a schema mapping sends a source arrow to a two-step path in a target schema. A production compiler and a small reference program both return the same normalized JSON. That agreement is encouraging, but it leaves several questions open. Did the reference program actually traverse the path, or did it copy the fixture's expected result? Did both programs reject an incomplete object map? Did they silently ignore an unknown field? Were migration-shaped documents evaluated or merely decoded? Which equality procedure justified a mapped path equation? Which input family was used when an associativity property passed?

Calling one program an "oracle" does not answer any of these questions. A test oracle is a role in a protocol, not an intrinsic property of a language. Haskell does not make a computation authoritative, and agreement between two programs does not make a theorem.

This paper therefore starts from an evidence boundary rather than a language preference. The accepted result is attached to:

- source commit `abd655f4db31f1e0a9e78de8bbe83c71805f46c7`;
- fixture capability set `slice-1`;
- a closed 24-case fixture corpus;
- seven fixed-seed QuickCheck properties;
- exact Haskell and Rust golden gates;
- an exact computed-output comparison; and
- a read-only review of the complete source range.

Within that boundary, the Haskell program is compact enough to audit and independent enough to catch implementation disagreement. Outside it, the program has no automatic authority. Its intended roles are semantic oracle, research compiler, and property-testing reference. It is neither a production dependency nor proof authority.

1.2 Research question

The broad P16 question is whether a compact functional model can serve as a semantic oracle for mappings and migrations. The accepted Slice 2 result answers the narrower question:

> Can an independently implemented Haskell program decode the shared CatDB fixture contract, compute the admitted finite schema, path, and mapping operations, reject selected invalid values, and agree exactly with a separate Rust implementation?

The bounded answer is yes. The migration half of the broader question remained outside Slice 2. At the accepted revision, migration and instance modules contained structural envelope types and canonical ordering only. Their fixtures were not semantic passes.

1.3 Contribution and prior-art boundary

Schemas as categories, instances as set-valued functors, schema mappings as functors, and the induced migration functors are established results [14, 20]. Relational encodings, composition, and executable categorical data integration also predate this work [18, 19, 21]. Schema mappings and data exchange have their own mature semantic literature [8, 9]. CQL already provides an executable categorical database language [6].

Nor is the validation method new in kind. QuickCheck introduced random property-based testing for Haskell [7]; SmallCheck made bounded exhaustive testing an explicit alternative [16]; and differential testing long predates this comparator [15].

The contribution reported here is consequently repository-specific:

1. a readable executable model of the selected CatDB schema, path, and mapping fragment;
2. a closed language-neutral fixture boundary with exact normalized outcomes and diagnostics;
3. an honest split between semantic computation and structural envelope handling;
4. deterministic, law-oriented bounded properties;
5. source-audited independence from the Rust implementation and fixture expected values;
6. two independent golden gates followed by a computed-output comparison; and
7. an immutable acceptance record with threats and nonclaims.

Evidence status. ACCEPTED COMPUTATIONAL for the exact Slice 2 result above; established categorical and testing foundations are attributed to prior work..

2 Evidence vocabulary and snapshot discipline

Definition 2.1 (Evidence labels)**.** This paper uses five labels:

ACCEPTED COMPUTATIONAL A gate passed at the reviewed source and corpus, and the result was internally accepted after independent review.

LIVE REPRODUCED The author pass reran an accepted command from an isolated extraction and observed the recorded result.

LATER EVIDENCE A result exists in a later repository revision but is not part of the accepted Slice 2 claim.

UNREVIEWED CANDIDATE A result exists only in the current working tree and has not passed the required terminal code-review and release-acceptance gate.

NONCLAIM The manuscript explicitly excludes the statement.

This distinction matters because the repository contains accepted Slice 4 code and a dirty Slice 5 candidate. Reading only current module exports would wrongly attribute instance validation, Δ migration, information-loss analysis, and schema diff to the earlier result. Conversely, reading only the historical commit would conceal later repairs and separately bounded extensions.

Table 1: Snapshot identities used in this paper.

Snapshot	Role	Result boundary
`377d383`	Parent research-contract baseline	Before Haskell Slice 2
`abd655f`	Reviewed and accepted source	24 fixtures; 16 semantic; 8 structural; 1,550 trials
`907914f`	Acceptance commit	Records review pass and internal acceptance
`0569158`	Later accepted Slice 4	33 fixtures; 29 semantic under `slice-4`; 4 structural; 2,350 trials
`e2f7cd7` plus dirty tree	Author-audit Slice 5 candidate	42 fixtures; analysis code and focused tests; uncommitted and unreviewed

Two roadmap files still label Slice 2/F2 as `OPEN`. Those labels predate the acceptance commit. This manuscript treats the dated release record and exact review range as the result authority and reports the stale roadmap text as an artifact-consistency defect. It does not edit the shared roadmaps or pretend that the discrepancy is absent.

3 Finite schema and path semantics

3.1 Profile representation

For the accepted profile, a schema is a finite presentation

$$S = (\mathsf{Ob}_S, \mathsf{Arr}_S, \mathsf{Attr}_S, \mathsf{Eq}_S).$$

Each generating arrow $f \in \mathsf{Arr}_S$ has declared source and target objects. Each attribute has a source object, scalar tag, and `required=true`. Each equation contains two serialized paths.

A path is a pair

$$p = (A, [f_1, \ldots, f_n]),$$

where A is the declared start. The empty list is the identity path at A. The Haskell representation uses closed algebraic data types and Aeson record codecs. The language is Haskell 2010 plus the package's declared GHC extensions and dependencies [1, 11].

Schema validation checks, in order relevant to this slice:

1. identifier syntax and profile-required attributes;
2. duplicate identifiers;
3. arrow and attribute endpoint resolution; and
4. typing and parallel endpoints for both sides of every equation.

Diagnostics contain a code, phase, RFC 6901 JSON Pointer, and sorted related semantic references [3]. Free-form host exceptions are not the comparison surface.

3.2 Path typing

The typing judgment

$$S \vdash p : A \rightsquigarrow B$$

is computed by a finite left-to-right traversal. The identity rule is

$$\frac{A \in \mathsf{Ob}_S}{S \vdash (A, []) : A \rightsquigarrow A}.$$

For a nonempty path, the validator resolves each named arrow and requires its source to equal the current endpoint. An unknown arrow or source mismatch produces a typed diagnostic rather than an empty result or exception.

For

$$S \vdash p : A \rightsquigarrow B, \qquad S \vdash q : B \rightsquigarrow C,$$

composition is represented in traversal order:

$$q \circ p = (A, \mathit{arrows}(p) +\!\!+ \mathit{arrows}(q)).$$

The function validates the schema and both paths before requiring the shared endpoint. Thus mathematical notation reads $q \circ p$, while serialized arrows list p first.

Example 3.1 (Accepted positive and negative coverage). The path fixtures include an identity, a valid chain, a valid chain composition, a source mismatch, and a composition endpoint mismatch. These examples cover both returned endpoints and exact diagnostics. They do not enumerate all graph shapes.

3.3 The accepted equality procedure

The target-path relation used during mapping validation is:

$$p \equiv_{1S} q \quad \Longleftrightarrow \quad p = q \ \vee \ (p, q) \in \mathsf{Eq}_S \ \vee \ (q, p) \in \mathsf{Eq}_S.$$

Only one exact declared equation is consulted. The relation is orientation-insensitive but is not closed under transitivity, congruence, context, or substitution.

Counterexample 3.2 (No transitive closure). If S declares $p = q$ and $q = r$ but not $p = r$, the accepted procedure need not conclude $p \equiv_{1S} r$. It is therefore not a complete equality decision for a finitely presented category.

This limit is central. A mapping accepted by the program preserves equations under $\equiv_1$, not under an arbitrary complete equational theory.
Evidence status. ACCEPTED COMPUTATIONAL for finite representation validation, path typing, composition, normalization, and direct equation checking at `abd655f`. General presented-category equality is NONCLAIM..

4 Mapping validation, identity, and composition

4.1 Closed mapping data

A mapping $F : S \to T$ has three finite total assignment arrays:

$$\begin{aligned} F_0 &: \mathsf{Ob}_S \to \mathsf{Ob}_T, \\ F_1 &: \mathsf{Arr}_S \to \mathsf{Path}_T, \\ F_a &: \mathsf{Attr}_S \to (\mathsf{Path}_T \times \mathsf{Attr}_T). \end{aligned}$$

An arrow image $F_1(f)$ must start at $F_0(\mathrm{src}(f))$ and end at $F_0(\mathrm{tgt}(f))$. An attribute image (p, a) first navigates a target path and then selects a target attribute. The path endpoint must be the attribute source, and the source and target scalar tags must match.

Validation rejects:

- a mismatched source or target schema ID;
- a missing or duplicate object, arrow, or attribute assignment;
- an unknown mapped object, arrow, or attribute;
- an arrow path with the wrong endpoints;
- an attribute expression with the wrong endpoint or scalar tag; and
- a source equation whose substituted sides are not related by $\equiv_{1T}$.

The arrays are normalized by source identifier. Duplicate evidence is rejected before normalization rather than collapsed.

4.2 Identity mapping

For a validated schema S, the constructor id_S emits:

$$\begin{aligned}(\mathrm{id}_S)_0(A) &= A,\\ (\mathrm{id}_S)_1(f : A \to B) &= (A, [f]),\\ (\mathrm{id}_S)_a(a \text{ at } A) &= ((A, []), a).\end{aligned}$$

The complete mapping is normalized and returned with a caller-supplied artifact identifier. Fixture equality includes that identifier. Law properties instead use an extensional comparator that ignores only `mapping_id`.

4.3 Composition by substitution

Let $F : S \to T$ and $G : T \to U$. The accepted implementation first validates both mappings and requires the target schema ID of F to equal the source schema ID of G.

For a T-path $p = (A, [f_1, \ldots, f_n])$, define bounded substitution:

$$G^*(p) = (G_0(A), \mathit{arrows}(G_1(f_1)) +\!\!+ \cdots +\!\!+ \mathit{arrows}(G_1(f_n)))\,.$$

Then:

$$\begin{aligned}(G \circ F)_0(A) &= G_0(F_0(A)),\\ (G \circ F)_1(f) &= G^*(F_1(f)).\end{aligned}$$

If $F_a(a) = (p, b)$ and $G_a(b) = (q, c)$, then

$$(G \circ F)_a(a) = \big(G^*(p)\,;q, c\big),$$

where ; appends the arrow list of q to the substituted prefix. The result is normalized and validated again against S and U.

Listing 1: Accepted path substitution core, lightly elided.

```
substitutePath :: Mapping -> Path -> Either Diagnostic Path
substitutePath mapping path = do
  start <- lookupObjectImage mapping (path_start path)
  imageArrows <-
    traverse
      (fmap path_arrows . lookupArrowImage mapping)
      (path_arrows path)
  pure (Path start (concat imageArrows))
```

Here `lookupObjectImage` returns the mapped object identifier, while `lookupArrowImage` returns a `Path`. Applying `path_arrows` inside the traversal therefore produces a list of arrow lists before `concat`; the displayed signature has no unused arrow-map parameter.

The implementation is finite and total in the Haskell sense that semantic failures are returned in `Either Diagnostic`. This does not imply termination or bounded resources for every arbitrarily large decoded input, nor does it establish categorical lawfulness beyond the implemented checks.

Evidence status. ACCEPTED COMPUTATIONAL for the closed mapping representation, selected validator, identity constructor, finite substitution, and composition..

5 Fixture semantics and honest structural boundaries

5.1 Closed documents

Every fixture contains:

- schema version `1.0.0`;
- stable fixture ID and one of eight categories;
- description and ordered assumptions;
- category-specific input; and
- one deterministic `ok`, `error`, or admissible `unsupported` expected outcome.

JSON object member order is not semantic, while normalized set-like arrays and path arrow arrays have explicit rules. The accepted program compares parsed JSON values, not source bytes. JSON syntax follows the shared interchange contract [2]; later schema-diff rules also refer to canonical JSON [17], although Slice 2 does not execute schema diff.

Closure is enforced in two ways. Generic record codecs set `rejectUnknownFields=True`, and manually decoded envelopes call explicit `closed` and `requireFields` checks. The decoder parses and re-encodes the category-specific input before rebuilding the complete fixture. Exact parsed-value equality between source and rebuilt fixture catches unknown or lossy fields.

The test suite constructs a path-typing fixture with an unknown field inside the nested schema. Successful decoding would fail the test.

5.2 Semantic versus structural cases

The Haskell evaluator dispatches by category. For one of the five admitted categories it computes a complete JSON outcome. For any of the other three it returns the internal marker `structural-only`. The case report then sets:

$$\begin{aligned} &\text{semantic:} && \mathsf{actual} = \mathsf{eval}(\text{input}), && \text{golden\_match} = (\mathsf{actual} = \mathsf{expected}), \\ &\text{structural:} && \mathsf{actual} = \texttt{null}, && \text{classification} = \texttt{structural-only}. \end{aligned}$$

The structural case still requires closed decoding and exact round trip. It does not compare a computed semantic result because none exists.

Counterexample 5.1 (An envelope is not an evaluator). The accepted Haskell type includes constructors named `Delta`, `Sigma`, and `Pi`. This permits a migration fixture to be decoded. At Slice 2, it does not imply that any of those operators is evaluated. Treating the four migration documents as semantic agreements would change the accepted count from 16 to 20 and would be false.

Evidence status. ACCEPTED COMPUTATIONAL for 24 closed round trips and exactly 16 semantic evaluations. Migration, information-loss, and diff semantics are NONCLAIM for Slice 2..

Table 2: Exact accepted Slice 2 classification.

Category	Cases	Classification	Operation
Schema parsing	3	semantic	Normalize or return exact parse/shape/type diagnostic
Path typing	3	semantic	Return start, endpoint, and arrow count, or diagnostic
Path composition	2	semantic	Return composed path or endpoint diagnostic
Identity mapping	2	semantic	Return canonical identity or schema diagnostic
Mapping composition	6	semantic	Return composed mapping or validation diagnostic
Migration validation	4	STRUCTURAL-ONLY	Closed round trip only; no result
Information-loss detection	2	STRUCTURAL-ONLY	Closed round trip only; no result
Schema diff	2	STRUCTURAL-ONLY	Closed round trip only; no result
Total	24	16 semantic, 8 structural	

6 The two-implementation comparison

6.1 Data flow

Figure 1 shows the accepted protocol. The expected outcome is a gate for each implementation, not an input to the Haskell evaluator. Only computed actual values cross the final semantic comparison.

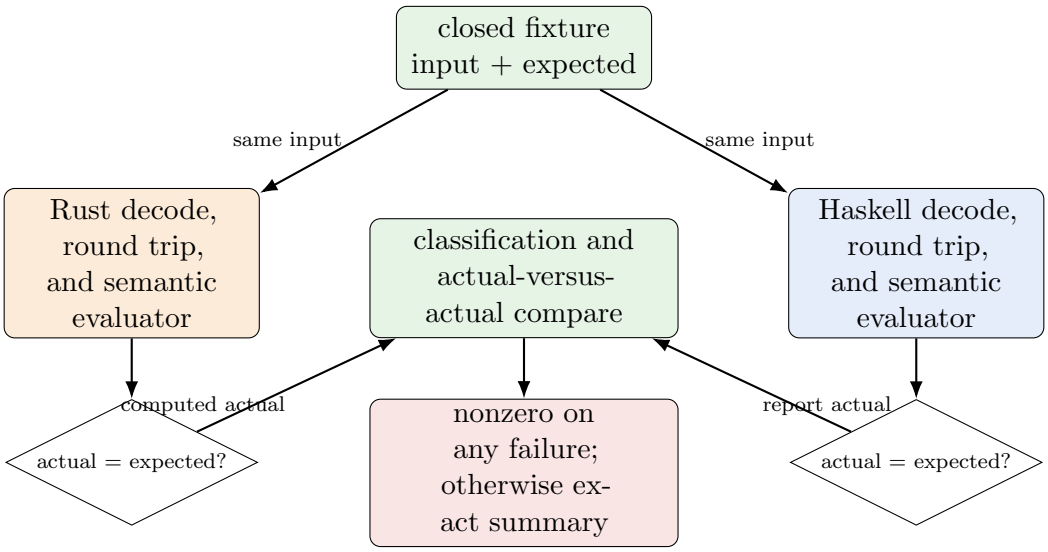

Figure 1: Accepted comparator data flow. Both implementations share the written fixture and golden value, but compute through separate source paths.

The Rust side decodes and round-trips the fixture, computes its actual outcome, and aborts on a golden mismatch. It then launches the selected Cabal program in the Haskell package directory, passes the selected GHC path, and requests a JSON report.

The report parser rejects:

- a failed or unstartable Haskell process;
- an unreadable or malformed report;
- wrong implementation or capability identity;
- a failed Haskell round trip or golden check;
- a missing or duplicate case;
- a category or classification difference;
- null semantic actuals or non-null structural actuals; and
- a Rust/Haskell computed-output difference.

For a structural case, both sides must classify it as structural and carry no semantic output. The comparator prints every disagreement and exits nonzero.

6.2 What independence means

The accepted source and review establish:

- no Rust dependency, FFI, process call, or Rust linkage in Haskell;
- no `unsafe` use in the Haskell tree;
- no dispatch by fixture ID;
- no use of `fixture_expected` inside the evaluator;
- semantic `report_actual` populated only from the Haskell computation; and
- a separate OS process and JSON report.

This is implementation independence, not specification independence. Both programs use the same fixture definition and expected output. Both may share a misreading, an incomplete corpus, a platform bug, or a flawed golden.

Remark 6.1 (Redundancy of the final comparison). Once Rust has proved only that its actual equals the fixture expected, and Haskell has proved only that its actual equals the same expected, equality of the two actuals follows. The final actual-versus-actual check is still useful defense against report association, transport, and comparator defects, but it does not add mathematical falsification power after both golden gates pass.

Evidence status. ACCEPTED COMPUTATIONAL for source-audited independent computation across the recorded process boundary. Independent specification and general implementation equivalence are NONCLAIM..

7 Law-oriented property tests

7.1 Seven accepted properties

Table 3: Deterministic accepted QuickCheck schedule.

Seed	Successful trials	Property
1729	250	Left and right identity for composed paths
1730	250	Piecewise path composition returns the full path and endpoint
1731	250	Three-segment path associativity
1732	200	Left and right mapping identity under extensional mapping equality
1733	200	Three-mapping substitution associativity under extensional equality
1734	200	One corrupted path source yields `PATH_SOURCE_MISMATCH`
1735	200	One omitted object assignment yields `INCOMPLETE_OBJECT_MAP`
	1,550	Total successful trials; no counterexample reported

The runner sets both `maxSuccess` and a replay generator for each property. GHC warnings are enabled with `-Wall -Wcompat`; the manual `strict` flag adds `-Werror`. The release command disables documentation so missing Haddock coverage is not confused with a source warning gate. GHC documents the warning-to-error behavior [10].

7.2 Generator boundary

The generator varies an integer size and constructs lawful values before testing a law. A generated schema is a chain:

$$E_0 \xrightarrow{a_0} E_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} E_n$$

with one required string attribute at E_0, no equations, and no branching. A generated mapping renames the corresponding entities, arrows, and attribute.

This construction avoids discarding mostly invalid random inputs and exercises composition over several finite lengths. It also creates a narrow sample space. In particular, generated properties do not cover:

- branching or cyclic schemas;
- multiple attributes or scalar tags;
- equation-bearing generated schemas;
- nontrivial attribute navigation;
- arbitrary lawful path images;
- simultaneous faults; or
- identifier punctuation at the canonical-order boundary.

The fixture corpus supplies some equation and error examples, but fixture examples do not widen the QuickCheck distribution.

Counterexample 7.1 (Passing trials are not universal quantification). An implementation can pass all 1,550 trials and still fail on a lawful branching schema, a two-equation target presentation, or an identifier containing punctuation. The accepted review found precisely such a latent ordering divergence for path keys containing `.` or `-`; the generated family did not expose it.

QuickCheck's contribution is automated random testing of stated properties, not proof [7]. The fixed schedule makes this run replayable under the recorded dependency version; it does not make the sample exhaustive. SmallCheck's explicit exhaustive-small-value model illustrates the distinction [16].

Evidence status. ACCEPTED COMPUTATIONAL for the exact seven properties, seeds, counts, and generators. Any universally quantified categorical law remains NONCLAIM..

8 Results

8.1 Accepted retained evidence

The retained Haskell report has SHA-256:

`d4725f04d8e3c908c3d11030936d4fdfeab9f174ed0b8b829f4c7c7316b33a6f`.

Programmatic inspection found that every semantic case had non-null `actual`, `round_trip=true`, and `golden_match=true`. Every structural case had `actual=null` and both Boolean gates true.

Table 4: Accepted Slice 2 result.

Measure	Result
Closed fixture round trips	24 / 24
Independently computed Haskell outcomes	16
Structural-only outcomes	8
Haskell golden mismatches	0
Failed Haskell cases	0
QuickCheck successful trials	1,550
QuickCheck counterexamples	0
Rust/Haskell semantic comparisons	16
Rust/Haskell disagreements	0
Independent review blockers / majors	0 / 0
Independent review verdict	PASS

The exact accepted comparator summary was:

```
summary left=rust right=haskell capability_set=slice-1 total=24 semantic-compared=16
    structural-compared=8 disagreements=0
```

The release record and review have SHA-256 values `db2405a7...e3d84` and `a9f8debc...c95`, respectively [5, 12].

8.2 Isolated historical rerun

The author pass extracted `abd655f` with `git archive`; it did not check out over the working tree. Cabal compilation and testing used a fresh build directory. With Cabal 3.16.1.0 and GHC 9.14.1, the following gates exited zero:

```
/opt/homebrew/bin/cabal check
/opt/homebrew/bin/cabal build all -fstrict --disable-documentation \
  --project-file=cabal.project \
  --with-compiler=/opt/homebrew/bin/ghc \
  --builddir=<fresh-directory>
/opt/homebrew/bin/cabal test all -fstrict --disable-documentation \
  --project-file=cabal.project \
  --with-compiler=/opt/homebrew/bin/ghc \
  --builddir=<same-directory> --test-show-details=direct
```

The test emitted the expected $250, 250, 250, 200, 200, 200, 200$ property counts and the 24/16/8/0 fixture summary. A standalone Haskell verification produced a report byte-identical to the accepted retained report. The exact historical Rust comparison also exited zero with the accepted summary.
Evidence status. LIVE REPRODUCED on 23 July 2026 for the exact historical source, fixture corpus, and observed dependency resolution..

8.3 Later accepted Slice 4, not retroactive evidence

Accepted Slice 4 source `0569158` adds finite instance validation and bounded pullback migration [13]. It also repairs or mitigates several Slice 2 review findings: that accepted tree has a Cabal freeze, explicit path-key equation ordering, owned temporary report files, per-comparison Cabal build isolation, and 11 properties.

Table 5: Accepted Slice 4 observations, not Slice 2 results.

Capability over accepted 33-case corpus	Semantic	Structural	Disagreements
`slice-1`	16	17	0
`slice-4`	29	4	0

The accepted Slice 4 strict Haskell tests completed 2,350 successful trials over 11 properties. Slice 4 makes all 13 migration cases semantic, including two finite Δ successes, validation errors, and exact unsupported Σ/Π results. Information loss and schema diff remain structural at that accepted boundary.

These observations demonstrate evolution, not a timeless equivalence theorem. They do not alter the accepted Slice 2 counts or make general migration available.

8.4 Current Slice 5 working-tree candidate

The author-audit working tree is based on `e2f7cd7` and contains uncommitted Slice 5 changes. Its Cabal package has been renamed from the historical `catdb-reference.cabal` manifest and `catdb-reference` package to `catdb-semantics.cabal` and `catdb-semantics`; the executable remains `catdb-reference`. The candidate adds `CatDB.Analysis` and nine fixtures—five information-loss fixtures and four schema-diff fixtures—expanding the corpus to seven information-loss cases and six schema-diff cases. It also adds a focused `loss-and-diff-properties` test suite.

The current corpus has 42 fixtures. Under `slice-1`, 16 are semantic and 26 are structural. Under `slice-4`, 29 are semantic and 13 are structural. Under `slice-5`, all 42 are semantic; 13 of these are information-loss and schema-diff cases. The full strict Haskell test command runs 13 properties and 2,750 trials. The focused candidate test runs two properties for 200 trials each and recomputes all 13 selected goldens.
Evidence status. UNREVIEWED CANDIDATE only. These observations are author-side verification, not an accepted Slice 5 result. They must not be promoted until terminal Claude code review and release acceptance..

9 Rust comparison and the role of Haskell

9.1 Different roles, shared contract

The Rust code is the production-oriented implementation for the admitted profile. It owns the shipping parser, diagnostic and normalization behavior, and later compiler/runtime boundaries. The Haskell package is a small readable calculation over the shared serialized profile. It serves as a semantic oracle, research compiler, and property-testing reference. Production deployment does not depend on it, and machine-checked claims belong to the separately scoped Lean formalization.

This division yields three useful failure classes:

1. both disagree with the fixture, suggesting two defects or a fixture issue;
2. one disagrees with the fixture and the other, localizing an implementation candidate; or
3. both agree with the fixture but the written definition or golden is wrong, a common-mode failure that comparison cannot detect.

A resolution must appeal to the written definition and recorded intent, not majority vote. Two programs agreeing against a definition do not redefine the definition silently.

9.2 Why Haskell is useful but not necessary

The accepted model benefits from ordinary functional-programming properties: small algebraic data types, explicit `Either` failures, pure finite transformations, and direct construction of law properties. These make the code convenient to inspect. They do not show that Haskell is uniquely suited to categorical database semantics. CQL's implementation history alone is enough to reject a language-necessity argument [6].

Nor does Haskell purity validate external effects. The comparison still depends on process launch, paths, environment variables, a temporary report, package resolution, files, and compilers. Backend queries, transactions, streaming, retries, and distributed faults are outside the Haskell program.

Evidence status. ACCEPTED COMPUTATIONAL that the Haskell source is an independently implemented finite reference for the selected operations. Haskell necessity, production deployment, and operational superiority are NONCLAIM..

10 Threat model and review findings

10.1 Threats addressed by the protocol

Table 6: Threats addressed at the accepted boundary.

Threat	Accepted control
Expected-value replay	Source and data-flow audit; expected used only for golden comparison
Rust reuse from Haskell	No Rust dependency, FFI, process use, or linkage in Haskell
Fixture-specific behavior	Category dispatch; no fixture-ID semantic branch
Unknown or lossy JSON	Closed typed codecs, exact parsed-value round trip, negative nested-field probe
Silent golden mismatch	Nonzero Rust and Haskell gates
Wrong report identity	Required implementation and capability fields
Missing or duplicate report case	Comparator rejection
Semantic/structural confusion	Required classification and nullability of `actual`
Computed-output difference	Exact parsed JSON comparison and nonzero exit
Tool selection ambiguity	`CATDB_CABAL`/`CATDB_GHC`; negative Cabal-path integration test

10.2 Accepted review findings

The independent review found no blocker or major issue. It retained five minor findings:

1. the temporary report name used process ID and nanosecond time rather than exclusive creation;
2. derived Haskell `Ord Path` could disagree with the written slash-separated path-key order for valid identifiers containing `.` or `-`;
3. the accepted package had no Cabal freeze or index-state pin;
4. package distribution metadata declared Apache-2.0 without a bundled package license file and omitted author/copyright metadata; and
5. property generators covered only a narrow linear family.

The review also noted inherited child stdout, a repository-root working directory assumption, declaration-order rather than written within-phase diagnostic tie-breaking, and a transient Cabal artifact race.

The first finding is a local attack surface. On a hostile shared temporary directory, an attacker who predicts and pre-creates the report path as a symlink may redirect the Haskell write. The accepted review judged the primary macOS per-user temporary directory mitigating but did not declare the design safe. Current Slice 4 code uses an owned temporary file; the historical result retains the finding.

The second finding is a specification-conformance gap masked by common-mode agreement: both languages used tuple-like order for the accepted corpus. Identifiers in the corpus did

not exercise the separator boundary. This is a clear example of why zero cross-language disagreement is not full contract conformance.

The third finding concerns replay stability. Cabal freeze files record a dependency solution for a project [4]. The isolated historical rerun resolved and passed, but that does not guarantee the same package graph at a future index state. Current code has a freeze; accepted Slice 2 did not.

10.3 Residual threats

The protocol does not eliminate:

- common errors in definitions or goldens;
- omitted fixture shapes or generator distributions;
- compiler, library, package-index, platform, or hardware defects;
- compromised tools or dependencies;
- resource exhaustion from very large finite inputs;
- multi-fault diagnostic-order divergences;
- Unicode, numeric, locale, and JSON edges outside the corpus;
- errors hidden by normalization;
- lack of mutation coverage or generator-distribution measurement; or
- the absence of organizationally independent specification authors.

Evidence status. ACCEPTED COMPUTATIONAL after review with the five minor findings retained. The protocol is not a supply-chain, sandbox, or hostile-environment security result..

11 Falsification contract

The bounded result should be rejected or narrowed if any of these conditions is demonstrated at the accepted source and corpus:

1. strict build or test failure under the recorded resolved toolchain;
2. inability to regenerate the retained report from accepted source and fixtures;
3. data flow from fixture expected values into a computed Haskell actual;
4. Haskell invocation or linkage of Rust;
5. fixture-ID-specific semantic behavior;
6. a golden mismatch in any of the 16 admitted semantic cases;
7. a changed parsed-value round trip in any accepted fixture;

8. acceptance of the nested unknown-field probe;
9. a Rust/Haskell actual difference in an admitted semantic case;
10. a semantic output attached to an accepted structural-only case;
11. comparator acceptance of a missing, duplicate, or misidentified report case;
12. a reproducible counterexample in the exact accepted property schedule; or
13. evidence that the review did not cover the stated source range.

An example outside the declared profile does not change the historical run, but it falsifies a generalization. For example, the accepted path-key ordering finding does not change the 24 results; it does falsify a claim of contract conformance for every valid identifier.

12 Exact nonclaims

The following are part of the result, not small-print qualifications:

1. QuickCheck trials are not proofs.
2. Rust and Haskell are not proved generally equivalent.
3. Two implementations do not establish specification correctness.
4. Haskell is not required by categorical database semantics.
5. Haskell is not a production dependency or service in this result.
6. Slice 2 does not semantically validate instances.
7. Slice 2 does not implement Δ, Σ, or Π.
8. Later bounded Δ is not full migration semantics.
9. General Kan extensions, quotients, and infinite instances are absent.
10. Slice 2 does not execute information-loss analysis or schema diff.
11. Query syntax, denotation, rewriting, and planning are absent.
12. SQL generation and backend equivalence are untested.
13. SQLite, PostgreSQL, API, file, and warehouse connectors are absent.
14. Transactions, cutover, rollback, CDC, and online schema change are unmodeled.
15. Cost, latency, throughput, scale, and memory are unmeasured.
16. Distributed execution, consistency, availability, and fault tolerance are unmodeled.
17. Equality modulo arbitrary path equations is not decided.
18. The direct-equation check is not complete.

19. Nulls, optional attributes, partial functions, bag semantics, and richer typesides are not represented by the accepted fragment.
20. Fixture agreement is not kernel-checked evidence.
21. Compiler and dependency trust are not eliminated.
22. A passing golden does not establish intended business meaning.
23. The accepted temporary report path is not claimed safe in a hostile shared directory.
24. The accepted Haskell dependency graph is not fully pinned.
25. Diagnostic priority is not generally equivalent on multi-fault inputs.
26. Accepted equation ordering is not generally conformant outside the accepted identifiers.
27. Structural fixture categories are not semantic P16 results.
28. The reference model does not replace database, security, integration, or user evaluation.

13 Reproducibility contract

The accompanying `reproducibility.md` records exact commands, tools, temporary directories, artifact hashes, and manuscript-build inspection. The minimum semantic reproduction has four stages:

1. extract the exact accepted Git object without modifying the working tree;
2. build and test the Haskell package with strict warnings in an isolated Cabal directory;
3. run the standalone verifier and compare its complete JSON hash; and
4. run the exact Rust/Haskell comparator from the extracted repository root.

Reproduction should preserve raw stdout, stderr, exit codes, tool versions, resolved dependency information, report JSON, fixture manifest and case hashes, repository revision, and dirty-state record. A future extension should also record distribution or exhaustive-small-model coverage rather than reporting only successful trial counts.

The accepted package lacked a freeze. For historical reconstruction, the best available record combines the source bounds, GHC 9.14.1, retained build logs, and the byte-identical report reproduced on 23 July 2026. The later freeze belongs to Slice 4 and must not be inserted into the historical source while claiming a byte-identical build recipe. Current package-qualified commands use `catdb-semantics`; historical commands retain the package name present in the accepted source. The executable remains `catdb-reference`.

14 Conclusion

The useful part of this result is not the sentence “we wrote it in Haskell.” It is the chain of observable boundaries around the program.

At one reviewed source revision, the Haskell implementation decoded a closed 24-case corpus, round-tripped every case, computed 16 schema/path/mapping outcomes, and marked eight later-slice cases as structural-only. Its seven fixed-seed properties completed 1,550 trials. The Rust implementation computed the same admitted outcomes through separate source, both programs passed exact golden gates, and the comparator found no disagreement. The author pass reproduced the Haskell report byte-for-byte and reran the historical comparison. Independent review found no blocker or major issue.

The same evidence also says what did not happen. Slice 2 did not evaluate a migration, prove a law, decide general path equality, execute a query, contact a backend, measure performance, or model a distributed system. Later finite instance and Δ work is accepted but separate. Current information-loss and schema-diff work remains an unreviewed candidate. Review findings expose common-mode ordering, dependency pinning, generator breadth, package metadata, and temporary-file risks.

A bounded executable oracle earns trust by making these distinctions easy to audit. Its strongest claim is exact agreement inside a named profile, not correctness everywhere else.

References

[1] Aeson maintainers. aeson: Fast JSON parsing and encoding. Hackage package and API documentation, 2026. URL https://hackage.haskell.org/package/aeson/docs/Data-Aeson.html. Accessed 23 July 2026.

[2] Tim Bray. The javascript object notation (JSON) data interchange format. Technical Report RFC 8259, Internet Engineering Task Force, 2017. URL https://www.rfc-editor.org/rfc/rfc8259.

[3] Paul Bryan, Kris Zyp, and Mark Nottingham. Javascript object notation (JSON) pointer. Technical Report RFC 6901, Internet Engineering Task Force, 2013. URL https://www.rfc-editor.org/rfc/rfc6901.

[4] Cabal maintainers. Cabal user's guide: Package projects and freeze files. Official Cabal documentation, 2026. URL https://cabal.readthedocs.io/en/latest/cabal-project-description-file.html. Accessed 23 July 2026.

[5] CatDB Research independent review. Independent senior review: Catdb research slice 2. Read-only repository review artifact, 2026. VERDICT: PASS; SHA-256 a9f8debc972623accb4ae9cc8edcf5e353752e9bbdc438093890969192a77c95.

[6] CategoricalData project. Categorical query language (CQL) documentation and tutorial. Official project documentation, 2026. URL https://categoricaldata.net/CQL/. Accessed 23 July 2026.

[7] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279. ACM, 2000. doi: 10.1145/351240.351266. URL https://doi.org/10.1145/351240.351266.

[8] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005. doi: 10.1016/j.tcs.2004.10.033. URL https://doi.org/10.1016/j.tcs.2004.10.033.

[9] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang-Chiew Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM Transactions on Database Systems*, 30(4):994–1055, 2005. doi: 10.1145/1114244.1114249. URL https://doi.org/10.1145/1114244.1114249.

[10] Glasgow Haskell Compiler project. GHC user's guide: Warnings and sanity-checking. Official compiler documentation, 2026. URL https://ghc.gitlab.haskell.org/ghc/doc/users_guide/using-warnings.html. Accessed 23 July 2026.

[11] Haskell Community. Haskell 2010 language report. Technical report, Haskell Community, 2010. URL https://www.haskell.org/definition/haskell2010.pdf.

[12] Matthew Long and The YonedaAI Collaboration. Slice 2 release evidence: Independent Haskell reference. CatDB Research repository release record, 2026. Accepted source abd655f4db31f1e0a9e78de8bbe83c71805f46c7; release record SHA-256 db2405a744da9eaf4b9d029e99234a13dc10f233848d4b2bc1cf9ebdefae3d84.

[13] Matthew Long and The YonedaAI Collaboration. Slice 4 release: Finite instances and bounded delta. CatDB Research repository release record, 2026. Internally accepted at source 056915834fcc2f172a4a72efe3629f4f753e45cf; release record SHA-256 afe1f08544ffb80b9dadd017a721371292599b863a5e06d92a231f2c90599254; later evidence, not part of the accepted Slice 2 result.

[14] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer, New York, second edition, 1998. ISBN 978-0-387-98403-2.

[15] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998. URL https://www.cs.tufts.edu/comp/150FP/archive/bill-mckeeman/DifferentailTesting.pdf.

[16] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and lazy SmallCheck: Automatic exhaustive testing for small values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, pages 37–48. ACM, 2008. doi: 10.1145/1411286.1411292. URL https://doi.org/10.1145/1411286.1411292.

[17] Anders Rundgren, Brian Jordan, and Samuel Erdtman. JSON canonicalization scheme (JCS). Technical Report RFC 8785, Internet Engineering Task Force, 2020. URL https://www.rfc-editor.org/rfc/rfc8785.

[18] Patrick Schultz and Ryan Wisnesky. Algebraic data integration. *Journal of Functional Programming*, 27:e24, 2017. doi: 10.1017/S0956796817000168. URL https://doi.org/10.1017/S0956796817000168.

[19] Patrick Schultz, David I. Spivak, Christina Vasilakopoulou, and Ryan Wisnesky. Algebraic databases. *Theory and Applications of Categories*, 32(16):547–619, 2017. doi: 10.70930/tac/lf9k6awo. URL https://www.tac.mta.ca/tac/volumes/32/16/32-16abs.html.

[20] David I. Spivak. Functorial data migration. *Information and Computation*, 217:31–51, 2012. doi: 10.1016/j.ic.2012.05.001. URL https://doi.org/10.1016/j.ic.2012.05.001.

[21] David I. Spivak and Ryan Wisnesky. Relational foundations for functorial data migration. In *Proceedings of the 15th Symposium on Database Programming Languages*, pages 21–28. ACM, 2015. doi: 10.1145/2815072.2815075. URL https://doi.org/10.1145/2815072.2815075.