OPERATION KOMOREBI

REALIZATION CALCULUS

PAPER III

Projector Resolved Seed Calculus III

Typed transfer, nonlinear packets, and obstruction theory

Read the exact source declarations, follow the map directions, and follow the current kernel verification and the preserved finite theorem map.

Komorebi P.

Lean and evidence guide • Release 1.0.0 / 20 October 2026

How to use this volume

A practical guide to the current release and the complete 41-row finite algebraic contract. Read source-exact proof excerpts, follow the direction of each map, distinguish exact coordinate transport from independent connection data, and locate every formal target.

Evidence at this edition. The release includes a fresh pinned compilation, a compiled ownership and axiom audit, upstream-only kernel replay, and adverse controls. The opening release chapter gives their exact scope. The preserved finite map still has 143 targets and its historical namespace audit has 468 entries. Those older counts describe the finite baseline, not the larger current closure.

Contents

The verified release 1

1 Reading the formal paper 4
- 1.1 Three levels of evidence . . . 4
- 1.2 A route through the source . . . 4
- 1.3 Notation that carries mathematical content . . . 5
- 1.4 What a short proof is doing . . . 5

2 Edges, words, and reachable packets 6
- 2.1 Graph orientation and finite enumeration . . . 6
- 2.2 Lists record chronological traversal . . . 7
- 2.3 From a transfer power to a finite walk sum . . . 8
- 2.4 Endpoint coefficients retain the empty path . . . 9
- 2.5 What the support theorem implies . . . 10

3 Heterogeneous products and exact coordinates 11
- 3.1 What varies in a heterogeneous path . . . 11
- 3.2 Arbitrary finite length on one packet type . . . 12
- 3.3 Declared support is a certificate . . . 12
- 3.4 Exact frames point from coordinates to ambient space . . . 13
- 3.5 Exact coordinate systems point the other way . . . 14
- 3.6 The naturality square is a typed equality . . . 15

4 Connections, defects, and holonomy 16
- 4.1 The minimal interface separates a choice from a theorem . . . 16
- 4.2 A counterexample to automatic flatness in the source . . . 16
- 4.3 Reversible connections use dependent fibers . . . 17
- 4.4 The identity between defect and holonomy . . . 18
- 4.5 Finite paths and their endpoint types . . . 19

5 Gauge reconstruction and packet obstruction blocks 20
- 5.1 The gauge action fixes its own convention . . . 20
- 5.2 Reconstructing a flat connection from canonical transport . . . 21
- 5.3 A packet resolution need not be orthogonal . . . 22
- 5.4 Extracting a witness supported in a packet . . . 23
- 5.5 When a fixed packet pattern survives a gauge . . . 24

6 Operation trees and certificates for individual candidates 25
6.1 The grammar does not assume algebraic laws . . . 25
6.2 The locality proof follows the syntax . . . 25
6.3 What the nonlinear extension adds outside this contract . . . 26
6.4 Residuals and a fixed proposed solution . . . 26
6.5 The quantifiers determine the conclusion . . . 27

7 The finite baseline: evidence, trust, and replay 28
7.1 The pinned environment is part of the claim . . . 28
7.2 A map gate and a namespace gate answer different questions . . . 28
7.3 Reading the inherited transcript . . . 29
7.4 The inherited Papers I–II references . . . 29
7.5 Replay from a preserved source copy . . . 30
7.6 Rebuilding this guide . . . 30
7.7 The edition's verification boundary . . . 30

8 The complete 41-row formal contract 32
8.1 Row 1: Packet graph and scalar packet matrix . . . 32
8.2 Row 2: Typed edge system, elementary edge operator, and finite transfer operator . . . 32
8.3 Row 3: Edge support outside the target-source block vanishes . . . 32
8.4 Row 4: Scalar packet multiplication and the two-edge composition dichotomy . . . 33
8.5 Row 5: Composable edge words and their operators . . . 33
8.6 Row 6: Broken and noncomposable words vanish . . . 33
8.7 Row 7: Recursive, tuple, explicit-word, and walk expansions . . . 33
8.8 Row 8: Successor laws for explicit words . . . 34
8.9 Row 9: Transfer powers equal recursive, explicit-word, and composable-walk expansions . . . 34
8.10 Row 10: Endpoint block and coefficient of a composable word . . . 34
8.11 Row 11: Fixed-endpoint coefficient sum . . . 35
8.12 Row 12: Fixed-length and global reachability obstructions . . . 35
8.13 Row 13: Fully typed two- and three-step packet expansions . . . 35
8.14 Row 14: Arbitrary-length heterogeneous factor expansion . . . 36
8.15 Row 15: Supported heterogeneous paths and vanishing . . . 36
8.16 Row 16: Canonical homogeneous linear coordinate transport . . . 36
8.17 Row 17: Exact self, cocycle, round-trip, and triangle laws . . . 36
8.18 Row 18: Exact finite path transport telescopes . . . 37
8.19 Row 19: Set-level exact coordinate transport . . . 37
8.20 Row 20: Representation, composition, and naturality of arbitrary functions . . . 38
8.21 Row 21: Minimal homogeneous noncanonical connection and its defects . . . 38
8.22 Row 22: Flat minimal connections have trivial defect and triangle holonomy . . . 38
8.23 Row 23: Explicit three-point connection with nontrivial loop . . . 39
8.24 Row 24: Heterogeneous reversible connection and gauges . . . 39
8.25 Row 25: Reversible cocycle, flatness, additive defect, holonomy, and obstruction . . . 39
8.26 Row 26: Defect-holonomy identity and equivalent flatness criteria . . . 40
8.27 Row 27: Canonical reversible frame connection is flat . . . 40
8.28 Row 28: Every flat reversible connection is gauge-equivalent to canonical transport . . . 40
8.29 Row 29: List-based reversible path transport is endpoint-only when flat . . . 41
8.30 Row 30: Indexed finite path transport and flat loops . . . 41
8.31 Row 31: Gauge action, covariance, and invariance . . . 41

8.32 Row 32: Finite complete idempotent packet resolution and blocks 42
8.33 Row 33: Exact block reconstruction, zero detection, and packet witnesses 42
8.34 Row 34: Triangle-obstruction block reconstruction and four-way detection 42
8.35 Row 35: Gauge conjugation and packet preservation 42
8.36 Row 36: Packet-preserving conjugation acts blockwise and preserves vanishing 43
8.37 Row 37: Gauge covariance of total and packetwise triangle obstructions 43
8.38 Row 38: Nonlinear operation-tree grammar, evaluation, and input support 43
8.39 Row 39: Operation-tree packet locality 44
8.40 Row 40: Candidate-specific solution and obstruction packet 44
8.41 Row 41: Candidate failure is equivalent to a local obstruction 44

A Monograph navigation 45

B Inherited declaration names 47

C Imports and exact dependency revisions 48

D Source and reproduction inventory 49

The verified release

Version 1.0.0 combines the finite algebraic development, exact nonlinear packet coordinates, and a new analytic packet interface. The original mathematical sources were restored in the exact pinned environment. The release then compiled the added packet modules, audited the compiled ownership boundary and transitive axioms, replayed the owned logical declarations into an environment initially containing only upstream dependencies, and exercised the rejection controls.

The mathematical objects are the actual ranges

The input space for a jet is the dependent finite product of the submodules ran P_{α_i}. The output is ran Q_γ. These submodules carry their inherited norms. The continuous jet uses the actual inclusions and output restriction. Its underlying multilinear map is the previously defined algebraic range jet.

$$\|J_{\gamma|\alpha}B\| \le \|Q_\gamma\|\,\|B\|.$$

The product of input-projector norms belongs to the estimate for the ambient extension. It is unnecessary in the sharper bound for the literal restricted jet. Closed ranges and the continuous coordinate equivalence are proved for continuous projectors.

Current source	Formal content
`PacketCoordinates.lean` `PacketDirectSum.lean`	Analysis and synthesis for a finite complete pairwise annihilating idempotent family, with range-valued coordinates and the direct-sum comparison.
`NonlinearPacketCoordinates.lean`	Exact coordinate representation of arbitrary functions, composition and coordinate changes.
`MultilinearPacketJets.lean`	Complete finite multilinear expansion, literal range jets, and reconstruction. Mixed input labels are retained.
`BoundedPacketJetNorm.lean`	The ambient continuous multilinear extension and its product norm bound.
`ContinuousPacketJets.lean`	Closed ranges, continuous packet coordinates, literal continuous range jets, the sharper norm bound, and specialization to `iteratedFDeriv`.
`PacketJetComposition.lean`	Composition over all finite ordered partitions and intermediate labels, structural vanishing, and one-way support inclusion.

All paths in this table are relative to `formal/KomorebiPaperIII/`. The formal sources state their own field, norm, finiteness and regularity assumptions. The composition theorem assumes `ContDiffAt K n f x` and `ContDiffAt K n g (f x)`. The intermediate and output projectors are continuous. Input range inclusions are continuous without an additional continuity hypothesis on the input family.

What the derivative notation commits to

Mathlib defines `iteratedFDeriv` as a total function. Reconstructing its chosen value is unconditional. That identity alone does not assert differentiability of an arbitrary map. The composition theorem supplies local differentiability hypotheses explicitly. Its `OrderedFinpartition n` uses the canonical order of the blocks by their greatest elements. Each original slot is selected by the partition embedding, and every intermediate label assignment is summed. In the real or complex smooth setting, symmetry identifies this formula with the manuscript's unordered-partition notation.

The support theorem says that a nonzero composite jet has at least one decorated partition with a nonzero outer jet and all inner jets nonzero. It gives no converse. Constituent maps can compose to zero, and distinct terms can cancel. The formal composition identity also covers order zero through the unique empty partition. The positive-order manuscript formula requires no new nullary convention.

The final evidence record

The final release compiled 16 production modules from an initially empty owned object directory. The compiled ownership audit recorded 608 owned declarations. The clean kernel replay accepted 594 logical declarations, with 14 generated runtime helpers separately inventoried. The full build, ownership audit, kernel replay, and control suite all passed.

The exact compiler is Lean `4.34.0-rc2`, commit `6a10ac8c22beadecabdbb0919c2b50214762f91d`. Mathlib is pinned at `aeb72f1e9981b6d35b426b16dca7759be3e7ba06`. The machine-readable summary is `formal/verification/FINAL_VERIFICATION_SUMMARY.json`. The full command transcripts and exact source snapshots accompany every phase. The owned audit examines compiled declarations by module ownership, including names outside the main namespace. It rejects project axioms, unreviewed opaque or partial declarations, unsafe declarations, and transitive axioms outside the standard basis `propext`, `Classical.choice`, and `Quot.sound`.

Generated runtime helpers are inventoried separately from the logical declarations replayed by the kernel. A helper count is not a theorem count. The rejection controls include foreign owned axioms, opaque, unsafe and partial declarations, false arithmetic, a proof placeholder, and a deliberately corrupted replay declaration. A harmless foreign namespace is a positive control. The negative controls pass when the designated defect is rejected for the recorded reason.

The original 41-row map below remains the finite baseline map. Its 143 targets comprise 88 theorem declarations and 55 constructions. The historical count of 468 namespace entries includes 229 theorem entries. These denominators are different from the current ownership closure and from the manuscript's 98 result environments plus two classical standard facts.

Reproduce the current result

The archive supplies owned source, pins, receipts and scripts. Restore the pinned Lean distribution and the dependency revisions using `formal/README.md`. Build in a separate working copy so the delivered manifest and historical transcripts stay intact. From `formal/`, run the four phases with a fresh object directory and unique labels:

```
python3 scripts/run_release_verification.py --lean /absolute/path/to/lean --phase full --label
 ↪  local_full --objects build/local_objects
python3 scripts/run_release_verification.py --lean /absolute/path/to/lean --phase audit --label
 ↪  local_audit --objects build/local_objects
```

```
python3 scripts/run_release_verification.py --lean /absolute/path/to/lean --phase replay --label
 ↪  local_replay --objects build/local_objects
python3 scripts/run_release_verification.py --lean /absolute/path/to/lean --phase controls --label
 ↪  local_controls --objects build/local_objects
```

The runner refuses a preexisting owned object directory for the full build. It records source hashes before and after compilation, the exact executable and command, output hashes, exit codes, and the expected diagnostics for adverse controls. Changing a dependency or toolchain creates a new verification event.

The remaining formal boundary

The manuscript's integral Taylor remainder, its factorial estimate, and the Stirling count of decorated terms are supplied as ordinary proofs. The 59 later results and two classical standard facts in Sections 18–27 were reviewed mathematically. Matching Lean formalizations for that expanded portion were not recovered. The source package preserves an October 16 visual claim of a larger closure as historical material, with its lack of matched source explicitly recorded.

These boundaries leave a complete mathematical reading edition with a precisely identified formal core and continuation. The statement register distinguishes fully mapped statements, partial formal correspondence, inherited references, and ordinary proofs. Public reuse terms remain an author decision, recorded separately from technical verification.

1 Reading the formal paper

Paper III turns three kinds of data into precise finite statements: directed packet operators, coordinate and connection transport, and local obstruction certificates. The preserved finite predecessor is small enough to read as a connected development. Its seven mathematical modules contain the authors' constructions and proofs. An aggregate import exposes them together. A separate audit module checks the compiled namespace. The opening release chapter records the enlarged current development.

This guide follows the definitions before using their theorem names. A declaration name is a precise destination, but its type determines what has actually been proved. The preserved finite map has 41 rows and 143 distinct targets. Of these, 88 are theorem declarations and 55 are constructions. A construction can be a structure, inductive type, definition, or abbreviation. The count does not mean 143 independent mathematical theorems.

1.1 Three levels of evidence

The preserved report, dated 3 September 2026, records a successful aggregate build of 3,076 jobs, the gate that checks parity with the theorem map, and a namespace audit of 468 declarations including 229 entries of kind `theoremInfo`. Paper III also preserves the kernel audit transcript. That transcript contains one axiom report for each of the 143 mapped targets and the final namespace PASS line.

For this edition, a Python reader reconciled all 41 map rows with their lexical source locations, all 143 explicit `#print axioms` directives, and all 143 transcript records. It checked the 16 files in the preserved formal component against the original checksum manifest. The theorem map and transcript hashes also match the values printed in the inherited verification report. These are fresh static checks of the preserved evidence. They do not replace a Lean execution; the opening release chapter separately records the current compiler and replay runs.

The expanded traditional companion includes additional ordinary mathematical proofs. The finite v1.0.1 formal contract stays fixed. A section of the final paper may combine a formally covered finite result with a stronger ordinary theorem. The presence of a related Lean name does not make every statement in that section part of the contract.

1.2 A route through the source

Module	Reading objective
`EdgeOperators`	Orientation, broken-word vanishing, and finite walk expansion.
`EdgeExtensions`	Endpoint blocks, reachability, and heterogeneous factor types.
`Transport`	Exact frames, the minimal connection, and a nontrivial loop.
`CoordinateNaturality`	Exact coordinate representation of arbitrary functions.
`ReversibleConnections`	Dependent fibers, defect, holonomy, gauges, paths, and blocks.
`HolonomyBlocks`	The commutation hypothesis for packetwise gauge covariance.
`NonlinearObstructions`	Tree locality and candidate-specific obstruction certificates.

The name of a file and the namespace of its declarations need not coincide. For example, the basic packet resolution declarations occur at the end of `ReversibleConnections.lean` under `Komorebi.PaperIII.HolonomyBlocks`. The complete map gives both the fully qualified name and the file and line. Follow that location when searching a source checkout.

1.3 Notation that carries mathematical content

Lean writes a linear map as `M →ₗ[R] N` and a linear equivalence as `M ≃ₗ[R] N`. The latter includes an inverse and its inverse laws. An arbitrary equivalence is `X ≃ Y`. A function `X → Y` carries no linear structure. Replacing one of these types with another changes the claim.

Bracketed arguments such as `[Semiring R]` are typeclass hypotheses. They supply operations and laws, not merely notation. Braced arguments such as `\{R : Type u\}` are implicit parameters inferred from use. `Type u` allows the object to live in a universe. Different universe names allow different input, fiber, and coefficient types. `autoImplicit false` requires undeclared identifiers to be rejected rather than silently generalized.

`Fintype V` supplies finite enumeration. `DecidableEq V` supplies a decision procedure for equality. Some definitions use `classical` so that a finite sum can be filtered by a proposition without demanding a decision procedure supplied by the user. This logical choice does not turn an abstract graph theorem into an efficient graph algorithm.

Three composition conventions must remain distinct:

Written expression	Execution order
`B * A`	Matrix A acts first, then B.
`g.comp f`	Linear map f acts first, then g.
`e.trans f`	Equivalence e acts first, then f.

A calculation translated from matrices to `LinearEquiv.trans` therefore reverses the written composition pattern. The underlying maps still execute in the same order.

1.4 What a short proof is doing

`rfl` closes a definitional equality: the two sides reduce to the same expression. `ext x` or `LinearEquiv.ext` reduces equality of maps to equality at an arbitrary input. `simp` uses definitions and registered simplification rules, including inverse laws and zero laws. `rw` rewrites by an explicit equality. `induction` exposes a finite recursive object, such as a list or tree, and proves the conclusion from the smaller objects.

The printed excerpts are exact source selections, not standalone files. Implicit variables, local instances, namespaces, and earlier lemmas may be outside a selection. Every exact UTF-8 excerpt has a source path, inclusive line range, and hashes in `generated/excerpts.json`. Copy the adjacent `.lean.txt` file when exact tokens matter. The PDF display uses font fallbacks for otherwise unsupported characters.

2 Edges, words, and reachable packets

2.1 Graph orientation and finite enumeration

`PacketGraph V E` has two fields, `source : E → V` and `target : E → V`. Distinct values of `E` may share endpoints, so parallel edges are allowed. The structure itself does not carry a finite enumeration. The theorems that sum over vertices or edges request the appropriate `Fintype` instances separately. This makes the boundary visible in each declaration.

The elementary scalar block is the matrix whose only potentially nonzero entry is in target row and source column.

Source: `KomorebiPaperIII/EdgeOperators.lean`.

```
def packetMatrix [Zero R] [DecidableEq V]
    (source target : V) (coefficient : R) : Matrix V V R :=
  fun row column ↦ if row = target ∧ column = source then coefficient else 0

@[simp] theorem packetMatrix_apply [Zero R] [DecidableEq V]
    (source target row column : V) (coefficient : R) :
    packetMatrix source target coefficient row column =
      if row = target ∧ column = source then coefficient else 0 :=
  rfl
```

For a column vector x, this block sends the source coordinate to the target coordinate and multiplies it by the stored coefficient. `EdgeSystem G R` is a coefficient function `E → R`. It adds weights to graph edges without changing their endpoints.

If $e : a \to b$ has coefficient r and $f : b \to c$ has coefficient s, the product for the traversal e followed by f is $E_f E_e$, and its coefficient is sr. A semiring need not be commutative. That coefficient cannot in general be replaced by rs.

Source: `KomorebiPaperIII/EdgeOperators.lean`.

```
theorem edgeOp_mul_edgeOp_of_composable
    [Fintype V] [DecidableEq V] [Semiring R]
    (e f : E) (h : G.target e = G.source f) :
    edgeOp G S f * edgeOp G S e =
      packetMatrix (G.source e) (G.target f)
        (S.coefficient f * S.coefficient e) := by
  ext row column
  by_cases hrow : row = G.target f
  · by_cases hcolumn : column = G.source e
    · simp [edgeOp, packetMatrix, Matrix.mul_apply, h, hrow, hcolumn]
    · simp [edgeOp, packetMatrix, Matrix.mul_apply, h, hcolumn]
  · simp [edgeOp, packetMatrix, Matrix.mul_apply, hrow]
```

The proof examines one matrix entry. The sum over intermediate vertices has at most one surviving term after support conditions are imposed. If the interface does not match, the companion theorem `edgeOp_mul_edgeOp_eq_zero` shows every intermediate term is zero. The zero conclusion depends on support, not on an assumed numerical cancellation.

2.2 Lists record chronological traversal

`Composable G` accepts the empty list and every singleton. A longer list must satisfy equality between the target of one edge and the source of the next at each step. The evaluator then places the first listed edge on the right:

Source: `KomorebiPaperIII/EdgeOperators.lean`.

```
def wordOp [Fintype V] [DecidableEq V] [Semiring R]
    (G : PacketGraph V E) (S : EdgeSystem G R) : List E → Matrix V V R
  | [] => 1
  | e :: rest => wordOp G S rest * edgeOp G S e

@[simp] theorem wordOp_nil [Fintype V] [DecidableEq V] [Semiring R] :
    wordOp G S [] = 1 :=
  rfl

@[simp] theorem wordOp_singleton [Fintype V] [DecidableEq V] [Semiring R]
    (e : E) : wordOp G S [e] = edgeOp G S e := by
  simp [wordOp]
```

Thus `[e,f,g]` evaluates to $E_g E_f E_e$. The empty list evaluates to the identity matrix, including when the finite vertex type is empty. The formal definitions cover the empty case directly.

The concatenation law gives a quick way to check the orientation:

Source: `KomorebiPaperIII/EdgeOperators.lean`.

```
theorem wordOp_append [Fintype V] [DecidableEq V] [Semiring R]
    (first second : List E) :
    wordOp G S (first ++ second) = wordOp G S second * wordOp G S first := by
  induction first with
  | nil => simp
  | cons e rest ih =>
      simp [wordOp, ih, Matrix.mul_assoc]
```

The operator for `first ++ second` is the operator for `second` multiplied by the operator for `first`. The proof is list induction plus associativity. Its order is essential in a noncommutative coefficient system.

The next theorem removes every word with a broken interface, not just a break at the start.

Source: `KomorebiPaperIII/EdgeOperators.lean`.

```
theorem wordOp_eq_zero_of_not_composable
    [Fintype V] [DecidableEq V] [Semiring R]
    (word : List E) (h : ¬ Composable G word) :
    wordOp G S word = 0 := by
  induction word with
  | nil => simp [Composable] at h
  | cons e rest ih =>
      cases rest with
      | nil => simp [Composable] at h
      | cons f tail =>
```

```
          simp only [Composable, not_and_or] at h
          rcases h with hbreak | htail
          · exact wordOp_eq_zero_of_first_break G S e f tail hbreak
          · change wordOp G S (f :: tail) * edgeOp G S e = 0
            rw [ih htail]
            simp
```

The induction distinguishes a mismatch in the first pair from a mismatch in the tail. The former invokes the support result for two edges. The latter uses the induction hypothesis to make the tail operator zero. Empty and singleton words cannot satisfy the noncomposability hypothesis, so those branches eliminate a contradiction.

2.3 From a transfer power to a finite walk sum

The transfer operator is $T = \sum_{e \in E} E_e$. Four related objects are kept separate:

1. `wordExpansion` recursively multiplies on the left by T.
2. `tupleWordOp` converts a tuple `Fin n → E` to an execution list.
3. `explicitWordExpansion` sums over every tuple of length n.
4. `walkExpansion` keeps precisely the composable tuples.

At length zero there is one empty tuple, not zero tuples. Its contribution is the identity. This is why the explicit sum agrees with T^0.

To pass from length n to length $n+1$, the proof separates a tuple into its first n entries and its final entry. `Fin.snocEquiv` supplies the finite indexing equivalence. Appending the final edge places its operator to the left of the prefix operator.

Source: `KomorebiPaperIII/EdgeOperators.lean`.

```
theorem explicitWordExpansion_succ
    [Fintype E] [Fintype V] [DecidableEq V] [Semiring R]
    (n : ℕ) :
    explicitWordExpansion G S (n + 1) =
      transferOp G S * explicitWordExpansion G S n := by
  unfold explicitWordExpansion
  rw [← (Fin.snocEquiv (fun _ : Fin (n + 1) ↦ E)).sum_comp]
  rw [Fintype.sum_prod_type]
  change (∑ final : E, ∑ prior : Fin n → E,
    tupleWordOp G S (n + 1) (Fin.snoc prior final)) = _
  simp only [tupleWordOp_snoc]
  unfold transferOp
  rw [Finset.sum_mul]
  simp_rw [Finset.mul_sum]
```

This is a finite distributivity argument. It uses neither an infinite geometric series nor a convergence limit. The proof then identifies the recursive expansion with the explicit tuple sum by induction on n.

Pruning the tuple sum uses the theorem for words with a broken interface, term by term:

Source: `KomorebiPaperIII/EdgeOperators.lean`.

```
theorem explicitWordExpansion_eq_walkExpansion
    [Fintype E] [Fintype V] [DecidableEq V] [Semiring R]
    (n : ℕ) :
    explicitWordExpansion G S n = walkExpansion G S n := by
  classical
  unfold explicitWordExpansion walkExpansion
  apply Finset.sum_congr rfl
  intro word _
  by_cases hwalk : Composable G (List.ofFn word)
  · simp [hwalk]
  · have hzero : tupleWordOp G S n word = 0 :=
      wordOp_eq_zero_of_not_composable G S (List.ofFn word) hwalk
    simp [hwalk, hzero]
```

When a tuple is composable, the filtered term is unchanged. When it is not, its unfiltered operator was already zero. Combining this equality with the recursive power identity gives `transferOp_pow_eq_walkExpansion`.

2.4 Endpoint coefficients retain the empty path

`reverseCoefficient` recursively multiplies coefficients in operator order. `terminalTarget` records the last endpoint of a nonempty word. `Connects G source target word` combines the endpoint conditions with composability. For an empty word it means `source = target`.

Source: `KomorebiPaperIII/EdgeExtensions.lean`.

```
theorem wordOp_apply_eq_if_connects
    [Fintype V] [DecidableEq V] [Semiring R]
    (word : List E) (source target : V) :
    wordOp G S word target source =
      if Connects G source target word then reverseCoefficient G S word else 0 := by
  cases word with
  | nil =>
      by_cases h : source = target
      · subst target
        simp [wordOp, Connects, reverseCoefficient]
      · have hreverse : target ≠ source := fun h' => h h'.symm
        simp [wordOp, Connects, h, hreverse]
  | cons first rest =>
      by_cases hwalk : Composable G (first :: rest)
      · rw [wordOp_eq_packetMatrix_of_composable G S first rest hwalk]
        simp [packetMatrix, Connects, hwalk, and_comm]
      · rw [wordOp_eq_zero_of_not_composable G S (first :: rest) hwalk]
        simp [Connects, hwalk]
```

This theorem includes both a nonempty composable word and the diagonal entry of the empty identity word. The endpoint sum is consequently valid for every natural length, including zero. The entry of T^n at the chosen endpoints is the sum of the reverse coefficients of exactly those words of length n connecting them.

2.5 What the support theorem implies

`Reachable G source target` means that some finite list connects the vertices. The empty list makes reachability reflexive. Absence of a walk of length n forces the corresponding entry of T^n to vanish. Absence of any walk gives this conclusion at every length.

The source also handles a seed supported on a specified set:

Source: `KomorebiPaperIII/EdgeExtensions.lean`.

```
theorem transferOp_pow_mulVec_apply_eq_zero_of_unreachable_support
    [Fintype E] [Fintype V] [DecidableEq V] [Semiring R]
    (n : ℕ) (x : V → R) (support : Set V) (target : V)
    (hx : ∀ source, source ∉ support → x source = 0)
    (hunreachable : ∀ source, source ∈ support → ¬ Reachable G source target) :
    ((transferOp G S ^ n) *ᵥ x) target = 0 := by
  rw [Matrix.mulVec_apply_eq_sum]
  apply Finset.sum_eq_zero
  intro source _
  by_cases hsource : source ∈ support
  · rw [transferOp_pow_apply_eq_zero_of_not_reachable G S n source target
      (hunreachable source hsource)]
    simp
  · rw [hx source hsource]
    simp
```

For each source vertex, the proof splits on membership in the support. Inside the support, the relevant transfer entry vanishes by unreachability. Outside it, the seed coordinate vanishes. Every term in the sum defining the matrix action is therefore zero.

The direction cannot be reversed without extra assumptions. A graph may contain a path whose coefficient is zero. Distinct paths may cancel over a ring. Thus a zero orbit coordinate does not by itself prove that the target is unreachable. The Lean theorem uses unreachability as a hypothesis and returns zero as a conclusion.

3 Heterogeneous products and exact coordinates

3.1 What varies in a heterogeneous path

There are two different kinds of heterogeneity. The matrix factor may change at every step while the packet index type stays fixed. Alternatively, the source, intermediate, and target packet types themselves may differ. The formal component includes a theorem for arbitrary finite length of the first kind and separate theorems for two and three steps of the second kind.

The statement for two steps displays all three packet types:

Source: `KomorebiPaperIII/EdgeExtensions.lean`.

```
theorem heterogeneousPacketPathExpansion_two
    {I0 : Type u0} {I1 : Type u1} {I2 : Type u2}
    [Fintype I1] [NonUnitalNonAssocSemiring R]
    (first : Matrix I1 I0 R) (second : Matrix I2 I1 R)
    (source : I0) (target : I2) :
    (second * first) target source =
      ∑ middle : I1, second target middle * first middle source := by
  rw [Matrix.mul_apply]
```

Only the intermediate type `I1` needs a finite enumeration. The source and target types are not enumerated in this entrywise identity. The coefficient hypothesis is `NonUnitalNonAssocSemiring R`. No multiplicative unit or associativity is needed for a single entry sum with two factors.

The statement for three steps introduces two finite intermediate types and a semiring:

Source: `KomorebiPaperIII/EdgeExtensions.lean`.

```
theorem heterogeneousPacketPathExpansion_three
    {I0 : Type u0} {I1 : Type u1} {I2 : Type u2} {I3 : Type u3}
    [Fintype I1] [Fintype I2] [Semiring R]
    (first : Matrix I1 I0 R) (second : Matrix I2 I1 R)
    (third : Matrix I3 I2 R) (source : I0) (target : I3) :
    ((third * second) * first) target source =
      ∑ middle2 : I2, ∑ middle1 : I1,
        (third target middle2 * second middle2 middle1) * first middle1 source := by
  rw [Matrix.mul_apply]
  simp_rw [Matrix.mul_apply, Finset.sum_mul]
  rw [Finset.sum_comm]
```

The parentheses on the coefficient product match `(third * second) * first`. The proof expands matrix multiplication twice, distributes multiplication across a finite sum, and exchanges the two finite sums. The packet index types remain separate throughout the calculation.

3.2 Arbitrary finite length on one packet type

`heterogeneousProduct` evaluates a list of square matrices in execution order. The head acts first and appears on the right. `heterogeneousPathSum` independently sums the intermediate packet at each recursive step. Its empty case is the identity coefficient, equal to one when source and target agree and zero otherwise.

Source: `KomorebiPaperIII/EdgeExtensions.lean`.

```
theorem heterogeneousPacketPathExpansion
    [Fintype V] [DecidableEq V] [Semiring R]
    (steps : List (Matrix V V R)) (source target : V) :
    heterogeneousProduct steps target source =
      heterogeneousPathSum steps source target := by
  induction steps generalizing source target with
  | nil =>
      by_cases h : source = target
      · subst target
        simp [heterogeneousProduct, heterogeneousPathSum]
      · have hreverse : target ≠ source := fun h' => h h'.symm
        simp [heterogeneousProduct, heterogeneousPathSum, hreverse]
  | cons step rest ih =>
      simp [heterogeneousProduct, heterogeneousPathSum, Matrix.mul_apply, ih]
```

The induction is on the list of factors. The base case is the matrix identity. The successor case is exactly the entry formula for matrix multiplication, with the induction hypothesis substituted into the remaining product. The theorem quantifies over every finite list, but all factors are `Matrix V V R`.

The traditional companion may treat a fully dependent chain with a new packet index type at every stage. That theorem lies outside the fixed Lean contract of 41 rows. Rows 13 and 14 make the distinction explicit: differing types at lengths two and three, varying factors on one common type at arbitrary finite length.

3.3 Declared support is a certificate

A `SupportedStep` stores an operator, a relation on source and target packets, and a proof that an entry is zero whenever the relation does not hold. The relation need not be the exact nonzero support. It can be a conservative relation of allowed packet transitions. `SupportedPath` says each successive step respects the corresponding relation.

Source: `KomorebiPaperIII/EdgeExtensions.lean`.

```
theorem heterogeneousPathSum_eq_zero_of_no_supportedPath
    [Fintype V] [DecidableEq V] [Semiring R]
    (steps : List (SupportedStep (V := V) (R := R))) (source target : V)
    (h : ¬ SupportedPath steps source target) :
    heterogeneousPathSum (steps.map SupportedStep.operator) source target = 0 := by
  induction steps generalizing source target with
  | nil =>
      have hreverse : target ≠ source := fun h' => h h'.symm
      simp [heterogeneousPathSum, hreverse]
  | cons step rest ih =>
```

```
      simp only [List.map_cons, heterogeneousPathSum]
      apply Finset.sum_eq_zero
      intro middle _
      by_cases hedge : step.relation source middle
      · have hrest : ¬ SupportedPath rest middle target := by
          intro hpath
          exact h ⟨middle, hedge, hpath⟩
        rw [ih middle target hrest]
        simp
      · rw [step.supported source middle hedge]
        simp
```

The proof splits at every candidate intermediate packet. If the first relation fails, the first matrix entry is zero by the stored certificate. If it holds, absence of an entire supported path forces absence of a supported tail, and the induction hypothesis makes the tail sum zero. This is useful even when exact nonzero support would be expensive to decide.

3.4 Exact frames point from coordinates to ambient space

In `Transport.lean`, a frame synthesizes an ambient vector from its coordinate representation. The homogeneous model uses an automorphism of one module, so source and target types look alike, but the direction still matters:

Source: `KomorebiPaperIII/Transport.lean`.

```
def exactTransport (frame : X → M ≃ₗ[R] M) (a b : X) : M ≃ₗ[R] M :=
  (frame a).trans (frame b).symm

@[simp] theorem exactTransport_apply (frame : X → M ≃ₗ[R] M)
    (a b : X) (x : M) :
    exactTransport frame a b x = (frame b).symm (frame a x) :=
  rfl
```

If Ψ_a is the frame, this is $\Psi_b^{-1}\Psi_a$. It first synthesizes through frame a, then analyzes through frame b. Cancellation of the intervening frame proves the cocycle law:

Source: `KomorebiPaperIII/Transport.lean`.

```
theorem exactTransport_cocycle (frame : X → M ≃ₗ[R] M) (a b c : X) :
    (exactTransport frame a b).trans (exactTransport frame b c) =
      exactTransport frame a c := by
  apply LinearEquiv.ext
  intro x
  simp [exactTransport]
```

The proof applies extensionality and simplifies an equivalence followed by its inverse. The results for self transport, return trips, triangles, and finite paths follow from the same cancellation. Exact transport obtained from one frame family cannot acquire a nontrivial loop merely by changing the path.

3.5 Exact coordinate systems point the other way

`CoordinateNaturality.lean` uses an equivalence `X ≃ Source` from an ambient type to a coordinate type. This is an analysis map, inverse in direction to the frame convention above.

Source: `KomorebiPaperIII/CoordinateNaturality.lean`.

```
def exactCoordinateTransport
    {X : Type u} {Source : Type v} {Target : Type w}
    (source : X ≃ Source) (target : X ≃ Target) : Source ≃ Target :=
  source.symm.trans target

@[simp] theorem exactCoordinateTransport_apply
    {X : Type u} {Source : Type v} {Target : Type w}
    (source : X ≃ Source) (target : X ≃ Target) (value : Source) :
    exactCoordinateTransport source target value = target (source.symm value) :=
  rfl
```

For analysis equivalences $\Phi_a : X \to A$ and $\Phi_b : X \to B$, the transport is $\Phi_b \Phi_a^{-1}$. Both conventions express the same operation of synthesizing and then analyzing. Check the direction of the supplied equivalence before comparing the formulas. This avoids an apparent inverse error.

> **Two directions for exact coordinates**
>
> A frame takes coordinates to ambient space. An analysis equivalence takes ambient data to coordinates. In both conventions, transport first returns to the ambient object and then enters the destination coordinates. The different written formulas reflect the different directions of the supplied equivalences.

An arbitrary function $f : X \to Y$ has coordinate representative

$$f_{a,r} = \Phi_r f \Phi_a^{-1}.$$

The type is an ordinary function `Source → Target`. It need not be linear, invertible, continuous, or differentiable. Composition is preserved exactly:

Source: `KomorebiPaperIII/CoordinateNaturality.lean`.

```
theorem exactCoordinateMap_comp
    {X : Type u} {Y : Type v} {Z : Type w}
    {CX : Type x} {CY : Type y} {CZ : Type z}
    (xCoordinate : X ≃ CX) (yCoordinate : Y ≃ CY)
    (zCoordinate : Z ≃ CZ) (first : X → Y) (second : Y → Z) :
    exactCoordinateMap xCoordinate zCoordinate (second ∘ first) =
      exactCoordinateMap yCoordinate zCoordinate second ∘
        exactCoordinateMap xCoordinate yCoordinate first := by
  funext value
  simp [exactCoordinateMap]
```

The middle coordinate analysis and synthesis cancel. The source proves equality of functions with `funext`, then simplifies at one value. No approximation is involved.

3.6 The naturality square is a typed equality

Source: `KomorebiPaperIII/CoordinateNaturality.lean`.

```
theorem exactCoordinateMap_naturality
    {X : Type u} {Y : Type v}
    {XA : Type w} {XB : Type x} {YR : Type y} {YS : Type z}
    (sourceA : X ≃ XA) (sourceB : X ≃ XB)
    (targetR : Y ≃ YR) (targetS : Y ≃ YS) (map : X → Y) :
    exactCoordinateTransport targetR targetS ∘
        exactCoordinateMap sourceA targetR map =
      exactCoordinateMap sourceB targetS map ∘
        exactCoordinateTransport sourceA sourceB := by
  funext value
  simp [exactCoordinateTransport, exactCoordinateMap, Function.comp_def]
```

On the left, represent the map in the first coordinate systems and then change target coordinates. On the right, change source coordinates first and then use the second representative. Both sides map `XA` to `YS`. Expanding them gives the same function.

This exact statement about sets already applies to a nonlinear function. It does not imply differentiability, a chain rule for jets, or convergence of a formal series. Those stronger properties require further structure and separate theorems. The formal contribution here is exact representation and naturality for arbitrary functions under exact equivalences.

4 Connections, defects, and holonomy

4.1 The minimal interface separates a choice from a theorem

The minimal `Connection R M X` stores an equivalence for each ordered pair of vertices, together with the identity law at equal endpoints.

Source: `KomorebiPaperIII/Transport.lean`.

```
structure Connection (R : Type u) (M : Type v) (X : Type w)
    [Semiring R] [AddCommMonoid M] [Module R M] where
  transport : X → X → M ≃ₗ[R] M
  transport_self : ∀ x, transport x x = LinearEquiv.refl R M
```

Neither reversal by inversion nor the cocycle equation is a field. Both could fail for an instance. This makes the distinction from exact frame transport substantive: exact transport has a prescribed construction that forces the cocycle law, while a connection carries separately chosen pair maps.

The minimal connection's defect is an equivalence. It compares the direct map with the route through an intermediate vertex by composing with an inverse:

Source: `KomorebiPaperIII/Transport.lean`.

```
def cocycleDefect (a b c : X) : M ≃ₗ[R] M :=
  (C.transport a c).symm.trans
    ((C.transport a b).trans (C.transport b c))
```

With ordinary function composition, this is

$$D_{a,b,c} = (\Gamma_{b,c}\Gamma_{a,b})\Gamma_{a,c}^{-1}.$$

It is based at the target coordinate space. In this multiplicative model, absence of a defect means that the comparison is the *identity*, rather than an additive zero map. Since the fibers here are homogeneous, the Lean type does not itself display which endpoint is the base. The definition fixes that interpretation.

4.2 A counterexample to automatic flatness in the source

`doubleEquiv` multiplies rational coordinates by two and divides by two for its inverse. The connection on three points chooses it for the directed edge $0 \to 1$ and chooses the identity for every other unequal pair.

Source: `KomorebiPaperIII/Transport.lean`.

```
def twistedThreePointConnection : Connection ℚ ℚ (Fin 3) where
  transport a b :=
    if a = b then LinearEquiv.refl ℚ ℚ
    else if a = (0 : Fin 3) ∧ b = (1 : Fin 3) then doubleEquiv
    else LinearEquiv.refl ℚ ℚ
  transport_self a := by simp
```

The loop $0 \to 1 \to 2 \to 0$ doubles the input one. Equality with the identity would therefore force $2 = 1$. The source proves nonidentity by applying the proposed equality of equivalences to that input:

Source: `KomorebiPaperIII/Transport.lean`.

```
theorem twisted_triangleHolonomy_ne_refl :
    twistedThreePointConnection.triangleHolonomy
        (0 : Fin 3) (1 : Fin 3) (2 : Fin 3) ≠ LinearEquiv.refl ℚ ℚ := by
  intro h
  have happly := congrArg (fun e : ℚ ≃ₗ[ℚ] ℚ ↦ e (1 : ℚ)) h
  norm_num [Connection.triangleHolonomy, twistedThreePointConnection,
    doubleEquiv] at happly
```

This is an example for the minimal connection interface. The reverse edge $1 \to 0$ is the identity, so it is not the inverse of doubling. The example must not be advertised as an instance of the stronger reversible interface.

4.3 Reversible connections use dependent fibers

The second interface assigns a module $F(a)$ to each vertex. Pair transport is a linear equivalence between the actual endpoint modules, and reverse transport is required to be its inverse.

Source: `KomorebiPaperIII/ReversibleConnections.lean`.

```
structure ReversibleConnection
    (R : Type u) (X : Type v) (F : X → Type w)
    [CommRing R] [∀ a, AddCommGroup (F a)] [∀ a, Module R (F a)] where
  transport : ∀ a b, F a ≃ₗ[R] F b
  transport_self : ∀ a, transport a a = LinearEquiv.refl R (F a)
  transport_reverse : ∀ a b, transport b a = (transport a b).symm
```

The hypotheses include a commutative ring, additive commutative groups on all fibers, and module structures. These permit subtraction of parallel linear maps. The cocycle law is still not a structural field. Thus reversible connections may have nontrivial triangle holonomy.

The additive defect has a source and target visible in its type:

Source: `KomorebiPaperIII/ReversibleConnections.lean`.

```
def cocycleDefect (a b c : X) : F a →ₗ[R] F c :=
  (C.transport b c).toLinearMap.comp (C.transport a b).toLinearMap -
    (C.transport a c).toLinearMap
```

Write

$$\Delta_{a,b,c} = \Gamma_{b,c}\Gamma_{a,b} - \Gamma_{a,c} \quad : F(a) \longrightarrow F(c).$$

The triangle holonomy is

$$H_{a,b,c} = \Gamma_{c,a}\Gamma_{b,c}\Gamma_{a,b} \quad : F(a) \longrightarrow F(a),$$

and the triangle obstruction is $O_{a,b,c} = H_{a,b,c} - I$. Unlike the minimal multiplicative defect, this Δ is a linear map that need not be invertible. Both APIs use the short name `cocycleDefect`, but they live in different namespaces and have different types.

4.4 The identity between defect and holonomy

Source: KomorebiPaperIII/ReversibleConnections.lean.

```
theorem triangleHolonomy_sub_id (a b c : X) :
    C.triangleObstruction a b c =
      (C.transport c a).toLinearMap.comp (C.cocycleDefect a b c) := by
  ext x
  simp only [triangleObstruction, triangleHolonomy, cocycleDefect,
    LinearMap.sub_apply, LinearMap.id_apply, LinearMap.comp_apply,
    LinearEquiv.coe_toLinearMap, LinearEquiv.trans_apply, map_sub]
  rw [C.transport_reverse a c]
  simp
```

Pointwise, the identity reads

$$(H - I)x = \Gamma_{c,a}(\Gamma_{b,c}\Gamma_{a,b}x - \Gamma_{a,c}x).$$

Linearity moves the subtraction through $\Gamma_{c,a}$. The reverse law makes $\Gamma_{c,a}\Gamma_{a,c}$ the identity. The source proof uses exactly these two steps after unfolding the definitions.

Because $\Gamma_{c,a}$ is injective, the obstruction is zero exactly when the defect is zero. Vanishing of the additive defect is the cocycle equation. Vanishing of holonomy minus identity means the holonomy is the identity. The theorem chain establishes these equivalences at a fixed ordered triple before quantifying over every triple.

Source: KomorebiPaperIII/ReversibleConnections.lean.

```
theorem flat_iff_triangleHolonomy_eq_refl :
    C.Flat ↔ ∀ a b c,
      C.triangleHolonomy a b c = LinearEquiv.refl R (F a) := by
  constructor
  · intro hflat a b c
    rw [← C.cocycleDefect_eq_zero_iff_triangleHolonomy_eq_refl a b c,
      C.cocycleDefect_eq_zero_iff_cocycleAt]
    exact hflat a b c
  · intro hhol a b c
    rw [← C.cocycleDefect_eq_zero_iff_cocycleAt,
      C.cocycleDefect_eq_zero_iff_triangleHolonomy_eq_refl]
    exact hhol a b c
```

Here `Flat` means the pair-cocycle law at every triple of vertices. It is a global algebraic condition on all pair transports. It is not a differential curvature condition obtained from a limiting process or a local chart cover.

4.5 Finite paths and their endpoint types

The path function on lists returns `F start ≃₁[R] F (pathEndpoint start vertices)`. The return type depends on the endpoint computed from the list. Flatness lets induction replace every remaining segment by direct transport, then collapse the next triangle by the cocycle law.

The indexed alternative packages the endpoints in the path's type. `VertexPath X a b` is a finite path from a to b. An empty path has equal endpoints. A constructor prefixes a step to a path from the intermediate vertex. Its transport theorem is:

Source: `KomorebiPaperIII/ReversibleConnections.lean`.

```
theorem indexedPathTransport_eq_direct_of_flat (hflat : C.Flat) :
    ∀ {a b : X} (path : VertexPath X a b),
      indexedPathTransport C path = C.transport a b := by
  intro a b path
  induction path with
  | nil a => exact (C.transport_self a).symm
  | cons a b c rest ih =>
      simp only [indexedPathTransport]
      rw [ih]
      exact hflat a b c
```

The loop corollary takes `path : VertexPath X a a`. Its output is then already an automorphism of `F a`, and the self law identifies it with the identity. This avoids an extra coercion between a computed endpoint and the initial vertex.

The paths are finite syntax. Nothing here states that an infinite composition exists or that approximating finite paths converge. The finite endpoint theorem is exact on its stated domain.

5 Gauge reconstruction and packet obstruction blocks

5.1 The gauge action fixes its own convention

A gauge is a family of fiber automorphisms $G_a : F(a) \to F(a)$. The source convention is

$$\Gamma^G_{a,b} = G_b \Gamma_{a,b} G_a^{-1}.$$

The definition has to prove that the new pair maps satisfy the self and reverse laws, because its return value is again a reversible connection.

Source: KomorebiPaperIII/ReversibleConnections.lean.

```
def gauge (G : Gauge (R := R) (F := F)) : ReversibleConnection R X F where
  transport a b := ((G a).symm.trans (C.transport a b)).trans (G b)
  transport_self a := by
    apply LinearEquiv.ext
    intro x
    rw [C.transport_self]
    simp
  transport_reverse a b := by
    rw [C.transport_reverse a b]
    apply LinearEquiv.ext
    intro x
    simp
```

The proof of each structural law is pointwise cancellation. Flatness is not needed to define the gauge action.

The defect has different source and target fibers, so it transforms with a gauge at each end:

Source: KomorebiPaperIII/ReversibleConnections.lean.

```
theorem gauge_cocycleDefect
    (G : Gauge (R := R) (F := F)) (a b c : X) :
    (C.gauge G).cocycleDefect a b c =
      (G c).toLinearMap.comp
        ((C.cocycleDefect a b c).comp (G a).symm.toLinearMap) := by
  ext x
  simp [gauge, cocycleDefect]
```

Thus $\Delta^G_{a,b,c} = G_c \Delta_{a,b,c} G_a^{-1}$. Holonomy is based at a, so both endpoint gauges are at that vertex:

Source: KomorebiPaperIII/ReversibleConnections.lean.

```
theorem gauge_triangleHolonomy
    (G : Gauge (R := R) (F := F)) (a b c : X) :
    (C.gauge G).triangleHolonomy a b c =
      ((G a).symm.trans (C.triangleHolonomy a b c)).trans (G a) := by
  apply LinearEquiv.ext
  intro x
  simp [gauge, triangleHolonomy]
```

The transformed matrix is conjugate to the original. Whether the holonomy is the identity is invariant, and the source also proves that the cocycle condition and flatness are invariant. A coordinate change may alter entries while preserving these exact statements.

5.2 Reconstructing a flat connection from canonical transport

Suppose a family of frames $\Psi_a : F(a) \simeq M$ and a base vertex o are supplied. The canonical connection is $K_{a,b} = \Psi_b^{-1}\Psi_a$. The source defines

$$G_a = \Gamma_{o,a} K_{a,o}.$$

Its type is a gauge on each fiber:

Source: `KomorebiPaperIII/ReversibleConnections.lean`.

```
def flatGauge (frame : ∀ a, F a ≃ₗ[R] M) (o : X) :
    Gauge (R := R) (F := F) :=
  fun a ↦ (canonicalTransport frame a o).trans (C.transport o a)
```

The reconstruction theorem proves $\Gamma_{a,b} = G_b K_{a,b} G_a^{-1}$ under flatness.

Source: `KomorebiPaperIII/ReversibleConnections.lean`.

```
theorem flat_isGaugeOfCanonical
    (hflat : C.Flat) (frame : ∀ a, F a ≃ₗ[R] M) (o a b : X) :
    C.transport a b =
      ((C.flatGauge frame o a).symm.trans
        ((canonicalConnection frame).transport a b)).trans
          (C.flatGauge frame o b) := by
  apply LinearEquiv.ext
  intro x
  have hg : (C.flatGauge frame o a).symm =
      (C.transport o a).symm.trans (canonicalTransport frame a o).symm := rfl
  rw [hg]
  have hcanonical_symm (p q : X) :
      (canonicalTransport frame p q).symm = canonicalTransport frame q p := rfl
  rw [hcanonical_symm]
  simp only [flatGauge, canonicalConnection, LinearEquiv.trans_apply]
  rw [show (C.transport o a).symm = C.transport a o by
    rw [C.transport_reverse o a]]
  simp only [canonicalTransport, LinearEquiv.trans_apply,
    LinearEquiv.symm_apply_apply, LinearEquiv.apply_symm_apply]
  have hao : (C.transport a o).trans (C.transport o b) = C.transport a b :=
    hflat a o b
  exact (congrArg (fun e : F a ≃ₗ[R] F b ↦ e x) hao).symm
```

The proof first expands the inverse of the defined gauge, then cancels the canonical frame maps. What remains is $\Gamma_{o,b}\Gamma_{a,o} = \Gamma_{a,b}$, the flatness equation at a, o, b.

The theorem is relative to supplied frames and a supplied vertex. It does not silently choose a vertex from an empty type. Nor does this declaration prove a classification theorem for topological bundles. Its result is an explicit algebraic formula for a flat normalized reversible pair connection on the supplied data.

5.3 A packet resolution need not be orthogonal

The formal interface for packet resolution asks for a finite complete family of idempotent endomorphisms.

Source: KomorebiPaperIII/ReversibleConnections.lean.

```
structure PacketResolution (R : Type u) (M : Type v) (I : Type w)
    [CommRing R] [AddCommGroup M] [Module R M] [Fintype I] where
  projector : I → M →ₗ[R] M
  complete : ∑ i, projector i = LinearMap.id
  idempotent : ∀ i, (projector i).comp (projector i) = projector i
```

Completeness is $\sum_i P_i = I$. Idempotence is $P_i^2 = P_i$. The structure does not assert pairwise orthogonality. The block from input packet ν to output packet μ is

$$A_{\mu\nu} = P_\mu A P_\nu.$$

Reconstruction only needs completeness on both sides:

Source: KomorebiPaperIII/ReversibleConnections.lean.

```
theorem sum_blocks (A : M →ₗ[R] M) :
    ∑ mu, ∑ nu, P.block A mu nu = A := by
  have hcomplete (x : M) : ∑ i, P.projector i x = x := by
    have hx := congrArg (fun L : M →ₗ[R] M ↦ L x) P.complete
    simpa using hx
  ext x
  simp only [block, LinearMap.sum_apply, LinearMap.comp_apply]
  calc
    (∑ mu, ∑ nu, P.projector mu (A (P.projector nu x))) =
        ∑ mu, P.projector mu (A (∑ nu, P.projector nu x)) := by
          apply Finset.sum_congr rfl
          intro mu _
          calc
            (∑ nu, P.projector mu (A (P.projector nu x))) =
                P.projector mu (∑ nu, A (P.projector nu x)) := by
                  exact (map_sum (P.projector mu)
                    (fun nu ↦ A (P.projector nu x)) Finset.univ).symm
            _ = P.projector mu (A (∑ nu, P.projector nu x)) := by
                  congr 1
                  exact (map_sum A
                    (fun nu ↦ P.projector nu x) Finset.univ).symm
    _ = ∑ mu, P.projector mu (A x) := by rw [hcomplete x]
    _ = A x := hcomplete (A x)
```

The proof applies the completeness equation to a vector. It combines the finite inner sum using linearity, applies completeness at the input, and then uses completeness at the output. No orthogonality cancellation appears.

All blocks vanish if and only if the map vanishes:

Source: KomorebiPaperIII/ReversibleConnections.lean.

```
theorem blocks_zero_iff (A : M →ₗ[R] M) :
    (∀ mu nu, P.block A mu nu = 0) ↔ A = 0 := by
  constructor
  · intro h
    rw [← P.sum_blocks A]
    simp [h]
  · intro h mu nu
    simp [block, h]
```

The forward direction rewrites the map as its double block sum. The reverse direction substitutes the zero map into the definition of a block. This result is a detector for the whole map even when the projectors overlap.

5.4 Extracting a witness supported in a packet

Idempotence is needed for the stronger witness statement:

Source: KomorebiPaperIII/ReversibleConnections.lean.

```
theorem exists_packetWitness_of_block_ne_zero
    (A : M →ₗ[R] M) (mu nu : I) (hne : P.block A mu nu ≠ 0) :
    ∃ x, P.projector nu x = x ∧ P.projector mu (A x) ≠ 0 := by
  classical
  have hexists : ∃ y, P.block A mu nu y ≠ 0 := by
    by_contra h
    apply hne
    ext y
    by_contra hy
    exact h ⟨y, hy⟩
  rcases hexists with ⟨y, hy⟩
  refine ⟨P.projector nu y, ?_, ?_⟩
  · have hid := congrArg (fun L : M →ₗ[R] M ↦ L y) (P.idempotent nu)
    simpa using hid
  · simpa [block] using hy
```

A nonzero block supplies some vector y on which that block is nonzero. The proof then takes $x = P_\nu y$. Idempotence ensures $P_\nu x = x$, and the original nonzero block value becomes $P_\mu A x \neq 0$.

The extraction is classical. It proves existence and does not supply a numerical search routine for an arbitrary module. Its input is a proof that a block is nonzero. It does not decide that predicate for the user.

Apply these results to $O = H - I$ on the base fiber of a fixed triangle. The declarations in rows 33 and 34 show

$$\Delta = 0 \quad \Longleftrightarrow \quad H = I \quad \Longleftrightarrow \quad O = 0 \quad \Longleftrightarrow \quad (\forall \mu, \nu)\ P_\mu O P_\nu = 0.$$

The theorem for the whole map, `obstructionDetection_equivalences`, records the three equivalences joining these four criteria. Nonzero obstruction blocks inherit the theorem giving a witness supported in a packet.

5.5 When a fixed packet pattern survives a gauge

Total obstruction always transforms by conjugation at the base vertex. For a *fixed* packet family to transform block by block, the gauge must commute with every projector.

Source: `KomorebiPaperIII/HolonomyBlocks.lean`.

```
def PacketPreserving (gauge : M ≃ₗ[R] M) : Prop :=
  ∀ packet, gauge.toLinearMap.comp (P.projector packet) =
    (P.projector packet).comp gauge.toLinearMap
```

The source first proves that the inverse gauge also commutes with the projectors. This lets the proof move both the gauge and its inverse past the output and input projectors:

Source: `KomorebiPaperIII/HolonomyBlocks.lean`.

```
theorem block_gauge_of_commutes
    (gauge : M ≃ₗ[R] M) (map : M →ₗ[R] M)
    (hpreserves : P.PacketPreserving gauge) (target source : I) :
    P.block (gaugeConjugate gauge map) target source =
      gaugeConjugate gauge (P.block map target source) := by
  ext value
  simp only [block, gaugeConjugate, LinearMap.comp_apply,
    LinearEquiv.coe_toLinearMap]
  have htarget (y : M) :
      P.projector target (gauge y) = gauge (P.projector target y) :=
    (LinearMap.congr_fun (hpreserves target) y).symm
  have hsource :
      gauge.symm (P.projector source value) =
        P.projector source (gauge.symm value) :=
    LinearMap.congr_fun
      (P.symm_commutes_of_packetPreserving gauge hpreserves source) value
  calc
    P.projector target
        (gauge (map (gauge.symm (P.projector source value)))) =
        gauge
          (P.projector target
            (map (gauge.symm (P.projector source value)))) :=
      htarget _
    _ = gauge
          (P.projector target
            (map (P.projector source (gauge.symm value)))) := by
      rw [hsource]
```

Consequently each block is conjugated separately, and each block's zero or nonzero status is invariant. Row 37 specializes this statement to a triangle obstruction and requires `P.PacketPreserving (G a)` at the base vertex.

An arbitrary gauge need not preserve the blocks of a fixed resolution. The unconditional fact is conjugation of the total obstruction. The conditional fact is conjugation of each fixed block. Keeping those hypotheses distinct is essential when interpreting a packet obstruction diagram.

6 Operation trees and certificates for individual candidates

6.1 The grammar does not assume algebraic laws

The nonlinear module starts with a finite syntax tree. Inputs, constants, unary labels, and binary labels can each have their own type.

Source: `KomorebiPaperIII/NonlinearObstructions.lean`.

```
inductive OperationTree
    (Input : Type u) (Constant : Type v) (Unary : Type w) (Binary : Type x)
    where
  | input : Input → OperationTree Input Constant Unary Binary
  | constant : Constant → OperationTree Input Constant Unary Binary
  | unary : Unary → OperationTree Input Constant Unary Binary →
      OperationTree Input Constant Unary Binary
  | binary : Binary → OperationTree Input Constant Unary Binary →
      OperationTree Input Constant Unary Binary →
      OperationTree Input Constant Unary Binary
```

To evaluate a tree, supply four functions: input values, constant values, unary semantics, and binary semantics. Unary and binary semantics are arbitrary functions on one common value type. They are not assumed additive, multilinear, differentiable, associative, or commutative. This generality is what makes the locality theorem applicable to any chosen semantics.

Every tree has finitely many nodes. The input type itself need not be finite. The function `inputs` computes a finite set of names appearing in the tree:

Source: `KomorebiPaperIII/NonlinearObstructions.lean`.

```
def inputs [DecidableEq Input] :
    OperationTree Input Constant Unary Binary → Finset Input
  | .input i => {i}
  | .constant _ => ∅
  | .unary _ child => inputs child
  | .binary _ left right => inputs left ∪ inputs right
```

A constant reads no input. A unary node reads exactly the names read by its child. A binary node reads the union of its two children's names. Because this is a set, repeated uses of one input are recorded once. The semantics may still depend nonlinearly on repeated occurrences.

6.2 The locality proof follows the syntax

Source: `KomorebiPaperIII/NonlinearObstructions.lean`.

```
theorem eval_congr_on_inputs [DecidableEq Input]
    (tree : OperationTree Input Constant Unary Binary)
    (first second : Input → Value)
    (constantValue : Constant → Value)
    (unaryValue : Unary → Value → Value)
    (binaryValue : Binary → Value → Value → Value)
```

```
    (hagree : ∀ i ∈ tree.inputs, first i = second i) :
    tree.eval first constantValue unaryValue binaryValue =
      tree.eval second constantValue unaryValue binaryValue := by
  induction tree with
  | input i =>
      exact hagree i (by simp [inputs])
  | constant c =>
      rfl
  | unary op child ih =>
      simp only [eval]
      congr 1
      exact ih fun i hi ↦ hagree i (by simpa [inputs] using hi)
  | binary op left right ihLeft ihRight =>
      simp only [eval]
      congr 1
      · exact ihLeft fun i hi ↦ hagree i (by simp [inputs, hi])
      · exact ihRight fun i hi ↦ hagree i (by simp [inputs, hi])
```

In an input case, the agreement hypothesis applies to that input's singleton support. In a constant case, both evaluations are definitionally equal. At a unary node, the induction hypothesis identifies the child values, and applying the same unary function preserves equality. At a binary node, the two child induction hypotheses apply because each child's support lies in the union.

The result states one implication: agreement on the syntactically named inputs gives equal evaluation. It does not claim every named input is semantically influential. For example, an operation may ignore one argument. The syntactic support remains a safe dependency set.

If `tree.inputs = ∅`, agreement on the empty set is vacuous, so the value is independent of the input environment. The source proves this as `eval_eq_of_inputs_eq_empty`. Constants and semantics are held fixed between the two evaluations. Changing the semantics would require a different comparison theorem.

6.3 What the nonlinear extension adds outside this contract

The traditional companion develops stronger ordinary statements about typed or colored trees, multilinear packet expansions, composition defects, and higher-order structures. The finite Lean contract contains the common-value unary/binary grammar and its input-locality theorem. It does not contain the dependent colored-tree development or a tree-level Faà di Bruno theorem.

This boundary is mathematical, not stylistic. A colored tree needs compatibility between node input and output types. A multilinear packet expansion needs algebraic hypotheses absent from arbitrary unary and binary semantics. The existing locality proof cannot certify these statements by a change of terminology.

6.4 Residuals and a fixed proposed solution

The obstruction API asks for a function `residual : Candidate → Packet → Residual`, with only `Zero Residual` assumed. A candidate is a solution when every packet residual is zero. A certificate stores one packet at which its residual is nonzero:

Source: `KomorebiPaperIII/NonlinearObstructions.lean`.

```
structure ObstructionPacket
    (residual : Candidate → Packet → Residual) (candidate : Candidate) where
  packet : Packet
  nonzero : residual candidate packet ≠ 0
```

The candidate appears in the certificate's type. A certificate for one candidate cannot directly be passed as a certificate for a different one. The theorem `ObstructionPacket.not_isSolution` uses the stored packet to contradict the condition that every packet residual is zero.

Classically, failure of the condition that every packet residual is zero supplies a nonempty certificate type. The combined equivalence is:

Source: `KomorebiPaperIII/NonlinearObstructions.lean`.

```
theorem not_isSolution_iff_nonempty_obstructionPacket
    {residual : Candidate → Packet → Residual} {candidate : Candidate} :
    ¬ IsSolution residual candidate ↔
      Nonempty (ObstructionPacket residual candidate) := by
  constructor
  · exact obstructionPacket_of_not_isSolution
  · rintro ⟨certificate⟩
    exact certificate.not_isSolution
```

The output is `Nonempty (ObstructionPacket residual candidate)`. This is a proposition asserting that the certificate type has an inhabitant. It is not a computable function that enumerates packets or finds one within a time bound.

Neither `Packet` nor `Candidate` needs a finite enumeration for this logical equivalence. The adjective "finite" in the overall contract describes the walk sums, paths, packet resolutions, and operation-tree syntax where finiteness is imposed. It should not be misread as an extra `Fintype` assumption on this candidate theorem.

6.5 The quantifiers determine the conclusion

For one proposed candidate c, the theorem says

$$\neg(\forall p,\ r(c,p) = 0) \quad \Longleftrightarrow \quad \text{a certificate of } r(c,p) \neq 0 \text{ exists.}$$

To show that a global solution does not exist, one needs the additional assertion that *every* candidate fails, or an argument producing an obstruction for each candidate. To prove existence of a global solution, one must produce a candidate with every residual zero. Neither conclusion follows merely from the certificate interface.

Once a witness is supplied, the interface makes local failure portable and checkable. The formal type records the residual, the candidate, the packet, and the exact nonzero assertion. A later theorem can build on these data without confusing rejection of one candidate with a universal impossibility result.

7 The finite baseline: evidence, trust, and replay

This chapter explains the preserved finite release and its historical gates. For the current combined build and replay, use the opening release chapter and `formal/README.md`. The old namespace counts in the excerpts apply only to that finite baseline.

7.1 The pinned environment is part of the claim

The formal project names Lean `leanprover/lean4:v4.34.0-rc2`, with reported Lean commit `6a10ac8c22beadecabdbb0919c2b50214762f91d`. Its Lake manifest pins Mathlib at `aeb72f1e9981b6d35b426b16dca7759be3e7ba06`. The project version is 1.0.1. Warnings are promoted to errors through `moreLeanArgs = ["-E", "warning"]`.

The preserved release deliberately omits compiled outputs and dependency caches. A fresh environment therefore needs the pinned toolchain and the dependencies named by the manifest. Rebuilding the PDF guide does not need Lean. Replaying the mathematical formalization does.

Descriptive dependency branches such as `master` or `main` are not substitutes for exact manifest revisions. Updating a dependency and obtaining a successful build may be useful, but it is a new verification event. The original event remains tied to the recorded environment.

7.2 A map gate and a namespace gate answer different questions

The authoritative `scripts/verify_theorem_map.py` checks the map row sequence, uniqueness, compiled declaration kinds, manuscript status annotations, inherited references, and explicit audit coverage. In the pinned compiled environment, it distinguishes the 88 theorem targets from the 55 construction targets. A lexical scan can locate these declarations in source, but does not independently elaborate their types.

The kernel audit file asks Lean to print transitive axioms for all mapped targets. It then inspects the compiled `Komorebi.PaperIII` namespace, including generated declarations and implementation auxiliaries. This second step protects the boundary beyond just the manually listed names.

Source: `KomorebiPaperIII/KernelAudit.lean`.

```
run_cmd do
  let env ← getEnv
  let mut audited : Nat := 0
  let mut theorems : Nat := 0
  for (name, info) in env.checked.get.constants.toList do
    if Name.isPrefixOf `Komorebi.PaperIII name then
      audited := audited + 1
      match info with
      | .thmInfo _ => theorems := theorems + 1
      | _ => pure ()
      if info.isUnsafe then
        throwError "unsafe Paper III declaration: {name}"
      match info with
      | .axiomInfo _ =>
          throwError "Paper III axiom declaration: {name}"
      | .opaqueInfo _ =>
          throwError "opaque Paper III declaration: {name}"
      | _ => pure ()
```

```
    let axioms ← Lean.collectAxioms name
    for axiomName in axioms do
      unless axiomName == ``propext || axiomName == ``Classical.choice ||
          axiomName == ``Quot.sound do
        throwError "unexpected axiom {axiomName} in Paper III declaration {name}"
if audited != 468 then
  throwError "Paper III namespace declaration count changed: expected 468, found {audited}"
if theorems != 229 then
  throwError "Paper III namespace theorem count changed: expected 229, found {theorems}"
logInfo m!"FAIL-CLOSED PAPER III v1.0.1 KERNEL AUDIT PASS: {audited} declarations ({theorems}
↪  theoremInfo); 143 explicit mapped targets; allowed axioms only: propext, Classical.choice,
↪  Quot.sound"
```

The gate rejects unsafe declarations, project axioms, opaque declarations, and any collected axiom outside `propext`, `Classical.choice`, and `Quot.sound`. It also checks the exact namespace and theorem-kind counts. These expected counts are version-specific: a legitimate addition requires revisiting them, not deleting the checks to make the build pass.

The source policy check additionally rejects proof placeholders and specified shortcuts. Text scanning alone cannot establish a theorem's correctness. It is one gate among the build, compiled map checks, axiom traversal, and checks of source integrity.

7.3 Reading the inherited transcript

The saved transcript has 143 named records. Some say that the target depends on no axioms. Others list a subset of the allowed standard basis. The final line records 468 audited namespace declarations, including 229 `theoremInfo` entries, and 143 explicit mapped targets.

The difference between 88 and 229 is not a discrepancy. Eighty-eight counts the theorem declarations deliberately mapped to the paper. The compiled namespace count includes generated theorems and other supporting theorem entries. Similarly, 143 and 468 refer to different denominators: the explicitly mapped targets versus every compiled entry under the audited namespace.

This guide checked that the transcript's named records agree with the map and the audit directives. It also checked the transcript hash against the inherited report. That establishes consistency and byte identity of the preserved materials. It does not authenticate an external compiler session by itself. A new execution is the direct way to regenerate compiler evidence from those sources.

7.4 The inherited Papers I–II references

Paper III records twelve unique declarations from the separately verified Papers I–II v1.2.1 bundle. They appear in fourteen manuscript references because two refinement declarations are used twice. The bundle is pinned by SHA-256:

`a20cf9353f0e882834d31e665dfd819a04e43fa606d402309be63e3b94447052`

The standalone Paper III project does not import or recompile those declarations. Its gate validates inherited annotations against `INHERITED_DEPENDENCY_MANIFEST_v1.2.1.md`. These references must therefore be described as inherited evidence. They are not extra entries in the 143-target Paper III namespace audit. Appendix B prints the exact dependency names and roles.

7.5 Replay from a preserved source copy

The source release offers an offline integrity mode. From its root, run:

```
bash VERIFY.sh
```

This mode checks release membership, checksums, map and audit consistency, manuscript status references, and the saved transcript. It does not run Lean. It is suitable for checking that a copied release still matches its own preserved manifest.

For a new formal run, work in a copy of the preserved release. From `formalization/lean`, run the release's documented commands in order:

```
lake --wfail build
python3 scripts/verify_theorem_map.py
lake env lean -E warning KomorebiPaperIII/KernelAudit.lean
```

Retain the exit code and the complete standard output and error stream for every command. The first command builds the project. The second checks the human map against the compiled environment and the original manuscript. The third regenerates the axiom reports and namespace result. A missing dependency, toolchain failure, timeout, or nonzero exit is not a passing proof run.

The release also supplies `bash VERIFY.sh --full`. That path creates a temporary copy, regenerates figures, runs Lean and its gates, rebuilds the original finite manuscript, and verifies the original artifact hashes. It requires its documented TeX, graphics, Python, and Lean dependencies. It verifies the preserved finite release, not the newly expanded companion PDF.

7.6 Rebuilding this guide

In the editable guide directory, `python3 build.py` runs XeLaTeX four times on the supplied source and generated data. It does not invoke Lean and does not regenerate data. This separation allows a typesetting replay without changing the preserved provenance tables.

To regenerate the guide's static data, run `build_guide_data.py --source PATH`, where `PATH` is the preserved finite release root containing `formalization/lean`, `kernel_audit_output.txt`, and `SHA256SUMS.txt`. In this package, supply `--source ../../checkpoint_v0.3.0/baseline/source/finite` and `--traditional ../../manuscript` from the guide directory. The old generator default names the historical layout. The generator uses only Python's standard library and never edits the source release.

The data generator writes the complete map, lexical source locations, import list, pins, exact excerpts, and static audit. The locator table for the traditional paper provides editorial navigation supplied by the adjacent traditional source directory or the explicit `--traditional` argument. Changing a locator does not alter the theorem map or extend the formal contract.

7.7 The edition's verification boundary

The finite portion of this guide supplies exact source excerpts, the complete 41-row map, the preserved transcript, and static reconciliation. The current release chapter separately reports fresh compiler and kernel verification. Its PDF is newly compiled and inspected. The package records separate computational checks and the paper's earlier bounded mathematical review. These records concern different objects. They do not establish fresh Lean proofs of every ordinary theorem.

The finite v1.0.1 contract has explicit types and evidence. To expand it, add the required types and proofs, update the map, run the pinned or newly documented environment, and retain the new transcript.

8 The complete 41-row formal contract

This index preserves every row, target, status, and boundary of the authoritative v1.0.1 map. Fully qualified declaration names are printed in full. File and line locations are a current lexical navigation aid, not a second compiled type audit. Each target appears in exactly one row. The 143 names comprise 88 theorem declarations and 55 constructions.

The finite limits are part of the statements: edge sums use finite packet and edge types, arbitrary-length heterogeneous products keep one common finite packet type, and packet reconstruction uses a finite complete idempotent family. Rows 38–41 have the separate grammar and candidate-specific scopes explained in Section 6.

8.1 Row 1: Packet graph and scalar packet matrix

Status and boundary. Defined; row is target and column is source

`Komorebi.PaperIII.EdgeOperators.PacketGraph`

structure, `KomorebiPaperIII/EdgeOperators.lean`, line 27.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.packetMatrix`

def, `KomorebiPaperIII/EdgeOperators.lean`, line 36.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.packetMatrix_apply`

theorem, `KomorebiPaperIII/EdgeOperators.lean`, line 40.

8.2 Row 2: Typed edge system, elementary edge operator, and finite transfer operator

Status and boundary. Defined for the finite scalar matrix model

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem`

structure, `KomorebiPaperIII/EdgeOperators.lean`, line 47.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.edgeOp`

def, `KomorebiPaperIII/EdgeOperators.lean`, line 55.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.edgeOp_apply`

theorem, `KomorebiPaperIII/EdgeOperators.lean`, line 58.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.transferOp`

def, `KomorebiPaperIII/EdgeOperators.lean`, line 162.

8.3 Row 3: Edge support outside the target-source block vanishes

Status and boundary. Proved

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.edgeOp_eq_zero_off_support`

theorem, `KomorebiPaperIII/EdgeOperators.lean`, line 65.

8.4 Row 4: Scalar packet multiplication and the two-edge composition dichotomy

Status and boundary. Proved; right factor acts first

`Komorebi.PaperIII.EdgeOperators.PacketGraph.packetMatrix_mul_packetMatrix`

theorem, `KomorebiPaperIII/EdgeExtensions.lean`, line 31.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.edgeOp_mul_edgeOp_eq_zero`

theorem, `KomorebiPaperIII/EdgeOperators.lean`, line 75.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.edgeOp_mul_edgeOp_of_composable`

theorem, `KomorebiPaperIII/EdgeOperators.lean`, line 92.

8.5 Row 5: Composable edge words and their operators

Status and boundary. Defined and proved

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.Composable`

def, `KomorebiPaperIII/EdgeOperators.lean`, line 106.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.wordOp`

def, `KomorebiPaperIII/EdgeOperators.lean`, line 112.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.wordOp_nil`

theorem, `KomorebiPaperIII/EdgeOperators.lean`, line 117.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.wordOp_singleton`

theorem, `KomorebiPaperIII/EdgeOperators.lean`, line 121.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.wordOp_append`

theorem, `KomorebiPaperIII/EdgeOperators.lean`, line 126.

8.6 Row 6: Broken and noncomposable words vanish

Status and boundary. Proved

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.wordOp_eq_zero_of_first_break`

theorem, `KomorebiPaperIII/EdgeOperators.lean`, line 135.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.wordOp_eq_zero_of_not_composable`

theorem, `KomorebiPaperIII/EdgeOperators.lean`, line 144.

8.7 Row 7: Recursive, tuple, explicit-word, and walk expansions

Status and boundary. Defined; all sums are finite

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.wordExpansion`

def, `KomorebiPaperIII/EdgeOperators.lean`, line 170.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.tupleWordOp`

def, `KomorebiPaperIII/EdgeOperators.lean`, line 177.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.explicitWordExpansion`

def, `KomorebiPaperIII/EdgeOperators.lean`, line 182.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.walkExpansion`

def, `KomorebiPaperIII/EdgeOperators.lean`, line 190.

8.8 Row 8: Successor laws for explicit words

Status and boundary. Proved

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.tupleWordOp_snoc`

theorem, `KomorebiPaperIII/EdgeOperators.lean`, line 198.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.explicitWordExpansion_succ`

theorem, `KomorebiPaperIII/EdgeOperators.lean`, line 210.

8.9 Row 9: Transfer powers equal recursive, explicit-word, and composable-walk expansions

Status and boundary. Proved

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.wordExpansion_eq_explicitWordExpansion`

theorem, `KomorebiPaperIII/EdgeOperators.lean`, line 226.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.explicitWordExpansion_eq_walkExpansion`

theorem, `KomorebiPaperIII/EdgeOperators.lean`, line 237.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.transferOp_pow_eq_wordExpansion`

theorem, `KomorebiPaperIII/EdgeOperators.lean`, line 252.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.transferOp_pow_eq_explicitWordExpansion`

theorem, `KomorebiPaperIII/EdgeOperators.lean`, line 263.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.transferOp_pow_eq_walkExpansion`

theorem, `KomorebiPaperIII/EdgeOperators.lean`, line 274.

8.10 Row 10: Endpoint block and coefficient of a composable word

Status and boundary. Defined and proved; coefficient order follows operator composition

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.reverseCoefficient`

def, `KomorebiPaperIII/EdgeExtensions.lean`, line 49.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.terminalTarget`

def, `KomorebiPaperIII/EdgeExtensions.lean`, line 54.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.Connects`

def, `KomorebiPaperIII/EdgeExtensions.lean`, line 59.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.wordOp_eq_packetMatrix_of_composable`

theorem, `KomorebiPaperIII/EdgeExtensions.lean`, line 75.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.wordOp_apply_eq_if_connects`

theorem, `KomorebiPaperIII/EdgeExtensions.lean`, line 98.

8.11 Row 11: Fixed-endpoint coefficient sum

Status and boundary. Defined and proved for fixed length

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.endpointWalkCoefficientSum`

def, `KomorebiPaperIII/EdgeExtensions.lean`, line 118.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.transferOp_pow_apply_eq_sum_walks`

theorem, `KomorebiPaperIII/EdgeExtensions.lean`, line 128.

8.12 Row 12: Fixed-length and global reachability obstructions

Status and boundary. Proved; reachability includes the empty path

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.transferOp_pow_apply_eq_zero_of_no_walk`

theorem, `KomorebiPaperIII/EdgeExtensions.lean`, line 142.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.Reachable`

def, `KomorebiPaperIII/EdgeExtensions.lean`, line 153.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.transferOp_pow_apply_eq_zero_of_not_reachable`

theorem, `KomorebiPaperIII/EdgeExtensions.lean`, line 157.

`Komorebi.PaperIII.EdgeOperators.PacketGraph.EdgeSystem.transferOp_pow_mulVec_apply_eq_zero_of_unreachable_support`

theorem, `KomorebiPaperIII/EdgeExtensions.lean`, line 169.

8.13 Row 13: Fully typed two- and three-step packet expansions

Status and boundary. Proved; intermediate packet types may differ

`Komorebi.PaperIII.EdgeOperators.heterogeneousPacketPathExpansion_two`

theorem, `KomorebiPaperIII/EdgeExtensions.lean`, line 195.

`Komorebi.PaperIII.EdgeOperators.heterogeneousPacketPathExpansion_three`

theorem, `KomorebiPaperIII/EdgeExtensions.lean`, line 205.

8.14 Row 14: Arbitrary-length heterogeneous factor expansion

Status and boundary. Defined and proved for varying factors on one common finite packet type

`Komorebi.PaperIII.EdgeOperators.heterogeneousProduct`

def, `KomorebiPaperIII/EdgeExtensions.lean`, line 221.

`Komorebi.PaperIII.EdgeOperators.heterogeneousPathSum`

def, `KomorebiPaperIII/EdgeExtensions.lean`, line 231.

`Komorebi.PaperIII.EdgeOperators.heterogeneousPacketPathExpansion`

theorem, `KomorebiPaperIII/EdgeExtensions.lean`, line 242.

8.15 Row 15: Supported heterogeneous paths and vanishing

Status and boundary. Defined and proved; support relations are explicit data

`Komorebi.PaperIII.EdgeOperators.SupportedStep`

structure, `KomorebiPaperIII/EdgeExtensions.lean`, line 258.

`Komorebi.PaperIII.EdgeOperators.SupportedPath`

def, `KomorebiPaperIII/EdgeExtensions.lean`, line 265.

`Komorebi.PaperIII.EdgeOperators.heterogeneousPathSum_eq_zero_of_no_supportedPath`

theorem, `KomorebiPaperIII/EdgeExtensions.lean`, line 273.

`Komorebi.PaperIII.EdgeOperators.supportedHeterogeneousPacketPathExpansion`

theorem, `KomorebiPaperIII/EdgeExtensions.lean`, line 296.

`Komorebi.PaperIII.EdgeOperators.heterogeneousProduct_apply_eq_zero_of_no_supportedPath`

theorem, `KomorebiPaperIII/EdgeExtensions.lean`, line 304.

8.16 Row 16: Canonical homogeneous linear coordinate transport

Status and boundary. Defined from one frame family

`Komorebi.PaperIII.Transport.exactTransport`

def, `KomorebiPaperIII/Transport.lean`, line 35.

`Komorebi.PaperIII.Transport.exactTransport_apply`

theorem, `KomorebiPaperIII/Transport.lean`, line 38.

8.17 Row 17: Exact self, cocycle, round-trip, and triangle laws

Status and boundary. Proved

`Komorebi.PaperIII.Transport.exactTransport_self`

theorem, `KomorebiPaperIII/Transport.lean`, line 44.

`Komorebi.PaperIII.Transport.exactTransport_cocycle`

theorem, `KomorebiPaperIII/Transport.lean`, line 51.

`Komorebi.PaperIII.Transport.exactTransport_roundTrip`

theorem, `KomorebiPaperIII/Transport.lean`, line 59.

`Komorebi.PaperIII.Transport.exactTransport_triangleLoop`

theorem, `KomorebiPaperIII/Transport.lean`, line 65.

8.18 Row 18: Exact finite path transport telescopes

Status and boundary. Defined and proved for arbitrary finite vertex lists

`Komorebi.PaperIII.Transport.pathEndpoint`

def, `KomorebiPaperIII/Transport.lean`, line 71.

`Komorebi.PaperIII.Transport.exactPathTransport`

def, `KomorebiPaperIII/Transport.lean`, line 76.

`Komorebi.PaperIII.Transport.exactPathTransport_eq_endpoint`

theorem, `KomorebiPaperIII/Transport.lean`, line 84.

`Komorebi.PaperIII.Transport.exactPathTransport_loop`

theorem, `KomorebiPaperIII/Transport.lean`, line 96.

8.19 Row 19: Set-level exact coordinate transport

Status and boundary. Defined and proved for arbitrary equivalences, without linearity assumptions

`Komorebi.PaperIII.Transport.exactCoordinateTransport`

def, `KomorebiPaperIII/CoordinateNaturality.lean`, line 24.

`Komorebi.PaperIII.Transport.exactCoordinateTransport_apply`

theorem, `KomorebiPaperIII/CoordinateNaturality.lean`, line 29.

`Komorebi.PaperIII.Transport.exactCoordinateTransport_self`

theorem, `KomorebiPaperIII/CoordinateNaturality.lean`, line 36.

`Komorebi.PaperIII.Transport.exactCoordinateTransport_symm`

theorem, `KomorebiPaperIII/CoordinateNaturality.lean`, line 43.

`Komorebi.PaperIII.Transport.exactCoordinateTransport_cocycle`

theorem, `KomorebiPaperIII/CoordinateNaturality.lean`, line 52.

`Komorebi.PaperIII.Transport.exactCoordinateTransport_triangle`

theorem, `KomorebiPaperIII/CoordinateNaturality.lean`, line 61.

8.20 Row 20: Representation, composition, and naturality of arbitrary functions

Status and boundary. Defined and proved at set level

`Komorebi.PaperIII.Transport.exactCoordinateMap`

def, `KomorebiPaperIII/CoordinateNaturality.lean`, line 71.

`Komorebi.PaperIII.Transport.exactCoordinateMap_apply`

theorem, `KomorebiPaperIII/CoordinateNaturality.lean`, line 77.

`Komorebi.PaperIII.Transport.exactCoordinateMap_id`

theorem, `KomorebiPaperIII/CoordinateNaturality.lean`, line 86.

`Komorebi.PaperIII.Transport.exactCoordinateMap_comp`

theorem, `KomorebiPaperIII/CoordinateNaturality.lean`, line 93.

`Komorebi.PaperIII.Transport.exactCoordinateMap_naturality`

theorem, `KomorebiPaperIII/CoordinateNaturality.lean`, line 109.

8.21 Row 21: Minimal homogeneous noncanonical connection and its defects

Status and boundary. Defined; only self-transport is structural

`Komorebi.PaperIII.Transport.Connection`

structure, `KomorebiPaperIII/Transport.lean`, line 113.

`Komorebi.PaperIII.Transport.Connection.CocycleAt`

def, `KomorebiPaperIII/Transport.lean`, line 123.

`Komorebi.PaperIII.Transport.Connection.Flat`

def, `KomorebiPaperIII/Transport.lean`, line 127.

`Komorebi.PaperIII.Transport.Connection.cocycleDefect`

def, `KomorebiPaperIII/Transport.lean`, line 133.

`Komorebi.PaperIII.Transport.Connection.triangleHolonomy`

def, `KomorebiPaperIII/Transport.lean`, line 138.

8.22 Row 22: Flat minimal connections have trivial defect and triangle holonomy

Status and boundary. Proved

`Komorebi.PaperIII.Transport.Connection.triangleHolonomy_eq_refl_of_flat`

theorem, `KomorebiPaperIII/Transport.lean`, line 142.

`Komorebi.PaperIII.Transport.Connection.cocycleDefect_eq_refl_of_flat`

theorem, `KomorebiPaperIII/Transport.lean`, line 147.

8.23 Row 23: Explicit three-point connection with nontrivial loop

Status and boundary. Constructed and proved over the rationals

`Komorebi.PaperIII.Transport.doubleEquiv`

def, `KomorebiPaperIII/Transport.lean`, line 160.

`Komorebi.PaperIII.Transport.doubleEquiv_apply`

theorem, `KomorebiPaperIII/Transport.lean`, line 170.

`Komorebi.PaperIII.Transport.twistedThreePointConnection`

def, `KomorebiPaperIII/Transport.lean`, line 177.

`Komorebi.PaperIII.Transport.twisted_triangleHolonomy_apply_one`

theorem, `KomorebiPaperIII/Transport.lean`, line 185.

`Komorebi.PaperIII.Transport.twisted_triangleHolonomy_ne_refl`

theorem, `KomorebiPaperIII/Transport.lean`, line 192.

8.24 Row 24: Heterogeneous reversible connection and gauges

Status and boundary. Defined over a commutative ring and a family of modules

`Komorebi.PaperIII.Transport.ReversibleConnection`

structure, `KomorebiPaperIII/ReversibleConnections.lean`, line 35.

`Komorebi.PaperIII.Transport.ReversibleConnection.Gauge`

abbrev, `KomorebiPaperIII/ReversibleConnections.lean`, line 49.

8.25 Row 25: Reversible cocycle, flatness, additive defect, holonomy, and obstruction

Status and boundary. Defined; theorem arguments are in traversal order

`Komorebi.PaperIII.Transport.ReversibleConnection.CocycleAt`

def, `KomorebiPaperIII/ReversibleConnections.lean`, line 52.

`Komorebi.PaperIII.Transport.ReversibleConnection.Flat`

def, `KomorebiPaperIII/ReversibleConnections.lean`, line 56.

`Komorebi.PaperIII.Transport.ReversibleConnection.cocycleDefect`

def, `KomorebiPaperIII/ReversibleConnections.lean`, line 62.

`Komorebi.PaperIII.Transport.ReversibleConnection.triangleHolonomy`

def, `KomorebiPaperIII/ReversibleConnections.lean`, line 67.

`Komorebi.PaperIII.Transport.ReversibleConnection.triangleObstruction`

def, `KomorebiPaperIII/ReversibleConnections.lean`, line 71.

8.26 Row 26: Defect-holonomy identity and equivalent flatness criteria

Status and boundary. Proved exactly

`Komorebi.PaperIII.Transport.ReversibleConnection.cocycleDefect_eq_zero_iff_cocycleAt`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 75.

`Komorebi.PaperIII.Transport.ReversibleConnection.triangleHolonomy_sub_id`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 96.

`Komorebi.PaperIII.Transport.ReversibleConnection.triangleObstruction_eq_zero_iff_cocycleDefect_eq_zero`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 107.

`Komorebi.PaperIII.Transport.ReversibleConnection.triangleObstruction_eq_zero_iff_triangleHolonomy_eq_refl`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 120.

`Komorebi.PaperIII.Transport.ReversibleConnection.cocycleDefect_eq_zero_iff_triangleHolonomy_eq_refl`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 139.

`Komorebi.PaperIII.Transport.ReversibleConnection.flat_iff_triangleHolonomy_eq_refl`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 146.

8.27 Row 27: Canonical reversible frame connection is flat

Status and boundary. Defined and proved

`Komorebi.PaperIII.Transport.ReversibleConnection.canonicalTransport`

def, `KomorebiPaperIII/ReversibleConnections.lean`, line 164.

`Komorebi.PaperIII.Transport.ReversibleConnection.canonicalConnection`

def, `KomorebiPaperIII/ReversibleConnections.lean`, line 171.

`Komorebi.PaperIII.Transport.ReversibleConnection.canonicalConnection_flat`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 181.

`Komorebi.PaperIII.Transport.ReversibleConnection.canonicalTransport_zero_connectionDefect`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 189.

`Komorebi.PaperIII.Transport.ReversibleConnection.canonicalTransport_triangleHolonomy_eq_refl`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 196.

8.28 Row 28: Every flat reversible connection is gauge-equivalent to canonical transport

Status and boundary. Gauge constructed and identity proved relative to a base vertex and frames

`Komorebi.PaperIII.Transport.ReversibleConnection.flatGauge`

def, `KomorebiPaperIII/ReversibleConnections.lean`, line 207.

`Komorebi.PaperIII.Transport.ReversibleConnection.flat_isGaugeOfCanonical`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 215.

8.29 Row 29: List-based reversible path transport is endpoint-only when flat

Status and boundary. Defined and proved for finite lists

`Komorebi.PaperIII.Transport.ReversibleConnection.pathTransport`

def, `KomorebiPaperIII/ReversibleConnections.lean`, line 241.

`Komorebi.PaperIII.Transport.ReversibleConnection.pathTransport_eq_direct_of_flat`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 248.

8.30 Row 30: Indexed finite path transport and flat loops

Status and boundary. Defined and proved for dependent fibers

`Komorebi.PaperIII.Transport.ReversibleConnection.VertexPath`

inductive, `KomorebiPaperIII/ReversibleConnections.lean`, line 262.

`Komorebi.PaperIII.Transport.ReversibleConnection.indexedPathTransport`

def, `KomorebiPaperIII/ReversibleConnections.lean`, line 267.

`Komorebi.PaperIII.Transport.ReversibleConnection.indexedPathTransport_eq_direct_of_flat`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 274.

`Komorebi.PaperIII.Transport.ReversibleConnection.indexedLoopTransport_eq_refl_of_flat`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 286.

8.31 Row 31: Gauge action, covariance, and invariance

Status and boundary. Defined and proved algebraically

`Komorebi.PaperIII.Transport.ReversibleConnection.gauge`

def, `KomorebiPaperIII/ReversibleConnections.lean`, line 294.

`Komorebi.PaperIII.Transport.ReversibleConnection.gauge_cocycleDefect`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 308.

`Komorebi.PaperIII.Transport.ReversibleConnection.gauge_triangleHolonomy`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 317.

`Komorebi.PaperIII.Transport.ReversibleConnection.gauge_cocycleAt_iff`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 326.

`Komorebi.PaperIII.Transport.ReversibleConnection.gauge_flat_iff`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 344.

`Komorebi.PaperIII.Transport.ReversibleConnection.gauge_triangleHolonomy_eq_refl_iff`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 353.

8.32 Row 32: Finite complete idempotent packet resolution and blocks

Status and boundary. Defined; orthogonality is not required

`Komorebi.PaperIII.HolonomyBlocks.PacketResolution`

structure, `KomorebiPaperIII/ReversibleConnections.lean`, line 379.

`Komorebi.PaperIII.HolonomyBlocks.PacketResolution.block`

def, `KomorebiPaperIII/ReversibleConnections.lean`, line 391.

8.33 Row 33: Exact block reconstruction, zero detection, and packet witnesses

Status and boundary. Proved; witness extraction is classical

`Komorebi.PaperIII.HolonomyBlocks.PacketResolution.sum_blocks`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 395.

`Komorebi.PaperIII.HolonomyBlocks.PacketResolution.blocks_zero_iff`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 420.

`Komorebi.PaperIII.HolonomyBlocks.PacketResolution.exists_packetWitness_of_block_ne_zero`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 430.

8.34 Row 34: Triangle-obstruction block reconstruction and four-way detection

Status and boundary. Proved for each fixed triangle and finite packet resolution

`Komorebi.PaperIII.HolonomyBlocks.obstruction_sum_blocks`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 452.

`Komorebi.PaperIII.HolonomyBlocks.obstructionBlocks_zero_iff`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 464.

`Komorebi.PaperIII.HolonomyBlocks.obstructionDetection_equivalences`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 479.

`Komorebi.PaperIII.HolonomyBlocks.exists_packetWitness_of_obstructionBlock_ne_zero`

theorem, `KomorebiPaperIII/ReversibleConnections.lean`, line 495.

8.35 Row 35: Gauge conjugation and packet preservation

Status and boundary. Defined

`Komorebi.PaperIII.HolonomyBlocks.PacketResolution.gaugeConjugate`

def, `KomorebiPaperIII/HolonomyBlocks.lean`, line 51.

`Komorebi.PaperIII.HolonomyBlocks.PacketResolution.PacketPreserving`

def, `KomorebiPaperIII/HolonomyBlocks.lean`, line 55.

8.36 Row 36: Packet-preserving conjugation acts blockwise and preserves vanishing

Status and boundary. Proved

`Komorebi.PaperIII.HolonomyBlocks.PacketResolution.symm_commutes_of_packetPreserving`

theorem, `KomorebiPaperIII/HolonomyBlocks.lean`, line 60.

`Komorebi.PaperIII.HolonomyBlocks.PacketResolution.block_gauge_of_commutes`

theorem, `KomorebiPaperIII/HolonomyBlocks.lean`, line 70.

`Komorebi.PaperIII.HolonomyBlocks.PacketResolution.gaugeConjugate_eq_zero_iff`

theorem, `KomorebiPaperIII/HolonomyBlocks.lean`, line 99.

`Komorebi.PaperIII.HolonomyBlocks.PacketResolution.block_gauge_eq_zero_iff`

theorem, `KomorebiPaperIII/HolonomyBlocks.lean`, line 111.

8.37 Row 37: Gauge covariance of total and packetwise triangle obstructions

Status and boundary. Proved; packetwise result assumes the base gauge commutes with every projector

`Komorebi.PaperIII.Transport.ReversibleConnection.gauge_triangleObstruction`

theorem, `KomorebiPaperIII/HolonomyBlocks.lean`, line 27.

`Komorebi.PaperIII.HolonomyBlocks.obstructionBlock_gauge_of_commutes`

theorem, `KomorebiPaperIII/HolonomyBlocks.lean`, line 131.

`Komorebi.PaperIII.HolonomyBlocks.obstructionBlock_gauge_eq_zero_iff`

theorem, `KomorebiPaperIII/HolonomyBlocks.lean`, line 145.

8.38 Row 38: Nonlinear operation-tree grammar, evaluation, and input support

Status and boundary. Defined for arbitrary unary and binary semantics

`Komorebi.PaperIII.NonlinearObstructions.OperationTree`

inductive, `KomorebiPaperIII/NonlinearObstructions.lean`, line 34.

`Komorebi.PaperIII.NonlinearObstructions.OperationTree.eval`

def, `KomorebiPaperIII/NonlinearObstructions.lean`, line 51.

`Komorebi.PaperIII.NonlinearObstructions.OperationTree.inputs`

def, `KomorebiPaperIII/NonlinearObstructions.lean`, line 66.

8.39 Row 39: Operation-tree packet locality

Status and boundary. Proved without algebraic assumptions on operations

`Komorebi.PaperIII.NonlinearObstructions.OperationTree.eval_congr_on_inputs`

theorem, `KomorebiPaperIII/NonlinearObstructions.lean`, line 77.

`Komorebi.PaperIII.NonlinearObstructions.OperationTree.eval_eq_of_inputs_eq_empty`

theorem, `KomorebiPaperIII/NonlinearObstructions.lean`, line 102.

8.40 Row 40: Candidate-specific solution and obstruction packet

Status and boundary. Defined

`Komorebi.PaperIII.NonlinearObstructions.IsSolution`

def, `KomorebiPaperIII/NonlinearObstructions.lean`, line 126.

`Komorebi.PaperIII.NonlinearObstructions.ObstructionPacket`

structure, `KomorebiPaperIII/NonlinearObstructions.lean`, line 130.

8.41 Row 41: Candidate failure is equivalent to a local obstruction

Status and boundary. Proved; extraction from nonsolution is classical and candidate-specific

`Komorebi.PaperIII.NonlinearObstructions.ObstructionPacket.not_isSolution`

theorem, `KomorebiPaperIII/NonlinearObstructions.lean`, line 136.

`Komorebi.PaperIII.NonlinearObstructions.obstructionPacket_of_not_isSolution`

theorem, `KomorebiPaperIII/NonlinearObstructions.lean`, line 144.

`Komorebi.PaperIII.NonlinearObstructions.not_isSolution_iff_nonempty_obstructionPacket`

theorem, `KomorebiPaperIII/NonlinearObstructions.lean`, line 156.

A Monograph navigation

The following locations point into the monograph companion. Printed pages differ from a PDF viewer's physical page index because the cover is unnumbered and the opening pages use Roman numerals. A locator connects related presentations; it does not certify every neighboring ordinary statement in Lean.

Canonical label	Final printed locator
`sec:purpose`	Section 1, p. 1
`sec:edge-operators`	Section 2, p. 2
`def:edge-system`	Definition 2.1, p. 2
`lem:edge-support`	Lemma 2.2, p. 2
`thm:edge-annihilation`	Theorem 2.3, p. 2
`thm:edge-composable`	Theorem 2.4, p. 3
`sec:words-walks`	Section 3, p. 3
`def:word-op`	Definition 3.1, p. 3
`thm:first-break`	Theorem 3.2, p. 3
`thm:nonwalk-vanish`	Theorem 3.3, p. 4
`def:recursive-word-expansion`	Definition 3.4, p. 4
`thm:power-recursive-expansion`	Theorem 3.5, p. 4
`lem:word-endpoint`	Lemma 3.7, p. 5
`thm:word-to-walk`	Theorem 3.8, p. 6
`cor:word-to-walk-entry`	Corollary 3.9, p. 6
`cor:fixed-reachability`	Corollary 3.10, p. 6
`cor:orbit-reachability`	Corollary 3.11, p. 6
`sec:canonical-transport`	Section 4, p. 7
`thm:exact-frame-transport`	Theorem 4.1, p. 7
`thm:exact-path-transport`	Theorem 4.2, p. 7
`def:canonical-J`	Definition 4.3, p. 8
`thm:J-cocycle`	Theorem 4.4, p. 8
`thm:canonical-naturality`	Theorem 4.5, p. 9
`cor:J-flat`	Corollary 4.6, p. 9
`sec:connection`	Section 5, p. 9
`def:minimal-connection`	Definition 5.1, p. 9
`thm:minimal-flatness`	Theorem 5.2, p. 10
`prop:twisted-loop`	Proposition 5.3, p. 10
`def:connection`	Definition 5.5, p. 11

Canonical label	Final printed locator
`def:calibration-flatness`	Definition 5.6, p. 12
`def:cocycle-defect`	Definition 5.8, p. 12
`def:triangle-holonomy`	Definition 5.9, p. 12
`thm:defect-holonomy`	Theorem 5.10, p. 13
`cor:canonical-zero-defect`	Corollary 5.11, p. 13
`def:path-transport`	Definition 5.12, p. 13
`prop:flat-endpoints`	Proposition 5.13, p. 14
`sec:gauge`	Section 6, p. 14
`def:gauge-change`	Definition 6.1, p. 14
`thm:gauge-covariance`	Theorem 6.2, p. 14
`thm:flat-gauge-canonical`	Theorem 6.3, p. 14
`sec:obstruction-packets`	Section 7, p. 15
`def:generic-obstruction`	Definition 7.1, p. 15
`thm:generic-obstruction`	Theorem 7.2, p. 15
`def:obstruction-packet`	Definition 7.4, p. 16
`thm:obstruction-detection`	Theorem 7.5, p. 17
`cor:packet-witness`	Corollary 7.6, p. 18
`prop:packet-gauge`	Proposition 7.7, p. 18
`sec:holonomy-walks`	Section 8, p. 19
`prop:heterogeneous-path`	Proposition 8.1, p. 19
`sec:example`	Section 9, p. 20
`sec:arbitrary-maps`	Section 10, p. 20
`prop:set-naturality`	Proposition 10.1, p. 20
`sec:operation-trees`	Section 11, p. 22
`def:compiled-operation-tree`	Definition 11.1, p. 22
`thm:tree-locality`	Theorem 11.2, p. 22
`def:signature`	Definition 11.4, p. 22
`def:operation-tree`	Definition 11.5, p. 23
`sec:ledger`	Section 12, p. 24

B Inherited declaration names

The following twelve names are inherited citations to the separately verified Papers I–II v1.2.1 bundle. The standalone Paper III build does not recompile them. Their fourteen manuscript uses are checked against the frozen manifest.

1. Reachability support hull

`Komorebi.PacketNaturalityCompletion.Reachability.algebraicOrbitHull_le_reachable_support`

2. Unreachable-packet obstruction

`Komorebi.PacketNaturalityCompletion.Reachability.unreachable_nonzero_packet_obstruction`

3. Nested coordinate refinement

`Komorebi.PaperII.NestedRefinement.refineCoordinates`

4. Refinement equivalence

`Komorebi.PaperII.NestedRefinement.coordinateRefinementEquiv`

5. Identity refinement

`Komorebi.PaperII.NestedRefinement.refineCoordinates_self`

6. Coarsen-after-refine identity

`Komorebi.PaperII.NestedRefinement.coarsenCoordinates_refineCoordinates`

7. Refine-after-coarsen identity

`Komorebi.PaperII.NestedRefinement.refineCoordinates_coarsenCoordinates`

8. Refinement cocycle and functoriality

`Komorebi.PaperII.NestedRefinement.refineCoordinates_comp`

9. Transitivity of refinement equivalences

`Komorebi.PaperII.NestedRefinement.coordinateRefinementEquiv_trans`

10. Coordinate-operator composition

`Komorebi.PaperII.NestedRefinement.coordinateOperator_comp`

11. Coordinate-operator naturality

`Komorebi.PaperII.NestedRefinement.coordinateOperator_naturality`

12. Prior packet-path power identity

`Komorebi.PacketNaturalityCompletion.Packets.transferBlock_pow_eq_packetPathSum`

C Imports and exact dependency revisions

The import table lists direct local file dependencies, not a theorem-level dependency graph. The adjacent JSON includes direct external imports as well. This appendix describes the preserved finite baseline. Its aggregate imports seven mathematical modules. The current release adds the continuation and packet calculus modules listed in the opening chapter.

Module	Direct local imports
`CoordinateNaturality`	External imports only.
`EdgeExtensions`	`EdgeOperators`
`EdgeOperators`	External imports only.
`HolonomyBlocks`	`ReversibleConnections`
`KernelAudit`	`KomorebiPaperIII`
`NonlinearObstructions`	External imports only.
`ReversibleConnections`	`Transport`
`Transport`	External imports only.
`KomorebiPaperIII`	`EdgeOperators`, `EdgeExtensions`, `Transport`, `ReversibleConnections`, `CoordinateNaturality`, `HolonomyBlocks`, `NonlinearObstructions`

The manifest's exact pinned revisions follow. They accompany the toolchain string `leanprover/lean4:v4.34.0-rc2`.

Package	Exact revision
mathlib	`aeb72f1e9981b6d35b426b16dca7759be3e7ba06`
plausible	`d9598f07b1bc701f1e3aae163d2681c1fd978793`
LeanSearchClient	`ba67e212be1197b84c1f1f6299488a10a3002713`
importGraph	`d8823026ac7ef130c253089d95685f9877b95323`
proofwidgets	`a8acbfd87375ff4abe14ce09db5b7664d383bc7f`
aesop	`18889deb9e83ea7420ef51c160d6f88552e744e3`
Qq	`507746ab8f4b643ccdacb2ec4cdb5853fa9f8ab3`
batteries	`4cac2177c37f5530c4da76aa8e4307f3fc9e4dcb`
Cli	`ab3a82db9fea14cf0fd7f5a2de650f4b534640af`

D Source and reproduction inventory

The preserved finite baseline component has nine Lean files, totaling 1,948 source lines, and sixteen files including the map, manifest, gate, pins, and documentation. The guide stores 44 exact excerpts. For each, `generated/excerpts.json` records the source file, inclusive line range, full-source hash, exact-excerpt hash, and PDF-display-file hash.

`generated/theorem_map.json` contains all 41 rows and complete source locations. `generated/mapped_source_index.json` provides an alphabetical index. `generated/inherited_axiom_records.json` records the 143 saved transcript reports. `generated/static_audit.json` separates current checks from inherited counts and records the checksums used. The original report and transcript remain in the preserved finite release. Current verification sources and receipts are under `formal/` in the release root.

`build.py` is a compile-only XeLaTeX builder. `build_guide_data.py` regenerates static data from an explicitly supplied preserved release. `verify_pdf.py` renders the PDF, prepares contact sheets, and records page-level layout checks. `BUILD_NOTES.txt` records authoring failures and their repairs, rather than silently treating failed attempts as successful verification.

The guide uses P052 for body text, Nimbus Sans for headings, DejaVu Sans Mono for code, and Latin Modern Math for mathematical symbols. Code displays use `fvextra`, preserving token order. Display fallbacks are documented separately from the exact UTF-8 source excerpts.