What theorems assume, prove, validate, and depend on?*

Nicolas Rouquette[1][0000−0003−3137−8690]

Jet Propulsion Laboratory, California Institute of Technology, CA 91189, USA
nfr@jpl.nasa.gov

Abstract. Proof assistants guarantee that individual theorems are correct, but provide no systematic way to understand how those theorems relate to other theorems and definitions they reason about. When a library contains hundreds of proofs, one may struggle to answer basic questions: Which theorems establish properties of this function? What assumptions does this proof rely on? Has every critical specification been verified? How does this proof relate to other proofs?
`doc-verification-bridge`, a documentation tool for Lean 4, automatically extracts and displays these relationships. For each theorem, it identifies other theorems it *depends on* and definitions the theorem *assumes* as preconditions, *proves* properties about, or *validates* (showing that a decidable function correctly implements its `Prop`-based specification). The tool performs static analysis to classify both definitions and theorems into three categories: statements about mathematical specifications (abstract properties expressed as logical propositions), statements about computational implementations (executable code), and bridging statements that connect the two (soundness and completeness proofs showing code correctly implements a specification).
The tool analyzes each theorem's signature to infer what it assumes, proves, and validates, and inspects proof terms to extract dependencies—all automatically, requiring no source code annotations. It generates navigable documentation where users can browse from a definition to all theorems about it, or from a theorem to everything it references. For verified software projects, this enables systematic tracking of which specifications have proofs and which remain unverified.

Keywords: Lean 4 · formal verification · documentation generation · verification coverage · proof engineering

1 Introduction

1.1 The Verification Coverage Gap

Modern proof assistants like Lean 4 provide extraordinary guarantees: every theorem in a formalized library has been mechanically verified against the system's axioms. Yet this guarantee addresses only *internal consistency*—it says nothing

about which theorems verify which specifications, what assumptions each proof relies on, or whether critical properties have been proven at all.

Consider a verified software library with 500 theorems. A developer asks: "Has the `lookup` function been proven correct?" To answer, one must manually search theorem statements for references to `lookup`, understand the proof structure, and hope naming conventions are consistent. The proof assistant guarantees each theorem is valid but provides no help answering practical questions about verification coverage.

This *verification coverage gap* becomes critical in safety-critical systems where regulatory requirements demand evidence that specific properties have been verified. Existing documentation tools generate API references but do not track the semantic relationships between theorems and the code they verify.

1.2 Contributions

This paper presents `doc-verification-bridge`, a documentation tool for Lean 4 that automatically extracts and displays verification relationships. Our contributions are:

1. A **four-category ontology** for classifying Lean declarations into mathematical specifications (Prop-based) and computational implementations (decidable), based on E.J. Lowe's metaphysical framework [1]. We show that Lean's support for the Curry-Howard isomorphism enables a clean realization of Lowe's categories that surpasses what is achievable with Description Logic-based approaches like UFO [10] or BFO [11].
2. **Automatic inference algorithms** that analyze theorem signatures to determine what each theorem *assumes*, *proves*, and *validates*—without requiring source code annotations.
3. **Proof dependency extraction** that inspects proof terms to identify which lemmas each theorem uses, revealing the verification dependency graph.
4. **Empirical evaluation** demonstrating the tool's effectiveness across multiple real-world Lean 4 libraries, including a refinement-based architecture description language where Mathlib's formalized mathematics directly supports software verification.

1.3 Paper Organization

Section 2 provides background on Lean 4. Section 3 presents our four-category ontology. Section 4 describes the inference algorithms. Section 5 details implementation. Section 6 presents empirical evaluation. Section 7 discusses related work, and Section 9 concludes. Appendix A provides a detailed case study.

2 Background and Motivation

2.1 Lean 4 and Dependent Type Theory

Lean 4 is a dependently-typed programming language and theorem prover based on the Calculus of Inductive Constructions [2]. Its type system implements the

Curry-Howard isomorphism: logical statements are types, proofs are terms, and the `Prop`/`Type` universe distinction separates propositions (proof-irrelevant) from data-carrying types.

This correspondence is foundational for our classification: a `def f : A → B` is simultaneously a function (computational) and a proof that `A` implies `B` (mathematical). The `Prop`/`Type` distinction directly determines mathematical versus computational classification.

A key distinction is between *propositions* and *decidable predicates*. A proposition `P : Prop` may or may not be provable, but provides no algorithm. A decidable predicate `decide p : Bool` computes a verdict and can optionally carry a proof that the verdict is correct. Verified software libraries frequently define both: a Prop-based *specification* and a Bool-returning *implementation*, then prove they correspond via *bridging theorems*.

2.2 The Specification-Implementation Divide

Consider a library that defines graphs and reachability. The specification might define transitive closure inductively:

```
inductive TransGen (r : α → α → Prop) : α → α → Prop
  | single : r a b → TransGen r a b
  | tail   : TransGen r a b → r b c → TransGen r a c
```

The implementation provides a decidable test:

```
def hasPath (g : Graph) (a b : Node) : Bool := ...
```

The verification goal is proving these correspond—a *bridging theorem*:

```
theorem hasPath_iff_transGen :
    hasPath g a b = true ↔ TransGen g.edge a b
```

Without tool support, discovering such relationships requires manual code inspection. Our tool automatically identifies `hasPath_iff_transGen` as a bridging theorem that *validates* `hasPath` against `TransGen`.

2.3 Existing Documentation Tools

doc-gen4 is the standard documentation generator for Lean 4 libraries. It produces navigable HTML showing declarations, types, and docstrings. However, it treats theorems as standalone entries without tracking their relationships to the definitions they verify.

Blueprint [4] supports structured proof development in Lean 4 with dependency graphs showing proof structure. It requires manual LaTeX annotations and focuses on tracking proof progress rather than specification-implementation relationships.

Neither tool answers: "Which theorems verify this function?" or "What specifications does this implementation satisfy?"

3 The Four-Category Ontology

3.1 Lowe's Metaphysical Framework

We base our classification on E.J. Lowe's *Four-Category Ontology* [1], a metaphysical framework distinguishing *universals* from *particulars* and *substantial* entities from *non-substantial* ones. Crucially, Lowe argues in Part I, Chapter 4 ("Formal Ontology and Logical Syntax") that his ontological categories correspond to the structure of formal logical systems—the very foundation upon which proof assistants are built.

Lowe's four categories are:

- **Kinds** (substantial universals): Abstract specifications that define *what things are*—the essential nature shared by instances.
- **Objects** (substantial particulars): Concrete instances that *instantiate* kinds—individual entities with identity.
- **Attributes** (non-substantial universals): Properties that entities *can possess*—characterizations independent of any particular bearer.
- **Modes** (non-substantial particulars): Property instances that *are possessed* by specific objects—concrete ways of being.

Lowe's insight is that logical syntax reflects ontological structure: predicates correspond to attributes (universals), terms to objects (particulars), and the instantiation relation connects them. This correspondence extends naturally to dependent type theory, where types serve as both logical propositions and data specifications.

3.2 Mapping to Lean Declarations

We adapt Lowe's framework to classify Lean declarations along two axes: *mathematical* (Prop-based, universal) versus *computational* (decidable, particular), and *substantial* (types) versus *non-substantial* (definitions).

The mapping to Lowe's categories:

Lowe's Category	**Mathematical**	**Computational**
Substantial (Kinds/Objects)	`mathematicalAbstraction`	`computationalDatatype`
Non-substantial (Attributes/Modes)	`mathematicalDefinition`	`computationalOperation`

- **Kinds** $\mapsto$ `mathematicalAbstraction`: The *kind* of all balanced binary search trees (a Prop-based specification defining what it means to be balanced).
- **Objects** $\mapsto$ `computationalDatatype`: A specific `RBMap` instance (a concrete data structure with runtime representation).
- **Attributes** $\mapsto$ `mathematicalDefinition`: The *attribute* of being sorted or well-formed (a Prop-returning predicate).
- **Modes** $\mapsto$ `computationalOperation`: The *mode* of a particular tree being balanced, witnessed by a decidable check (a Bool-returning function).

Types (inductive types, structures, classes) are classified as:

- `mathematicalAbstraction`: Prop-based types like `Chain`, `TransGen`, type classes defining algebraic structures
- `computationalDatatype`: Data-carrying types like `List`, `HashMap`, enumeration types with runtime representation

Definitions are classified by their return type:

- `mathematicalDefinition`: Returns `Prop`—predicates, relations, specifications (e.g., `IsAcyclic`, `WellFormed`)
- `computationalOperation`: Returns non-`Prop`—algorithms, accessors, decidable predicates (e.g., `lookup`, `hasPath : Bool`)

3.3 Mathematical vs Computational Distinctions

The tool automatically distinguishes mathematical from computational declarations using Lean's type system. The key insight: Prop-returning definitions describe *what should hold* (specification), while Bool-returning definitions describe *how to check* (implementation).

For a definition `def f (x : α) : β`, classification proceeds:

1. Telescope through parameters to reach the final return type
2. Apply weak-head normal form reduction
3. If the normalized type is `Prop`, classify as `mathematicalDefinition`
4. Otherwise, classify as `computationalOperation`

For theorems, we define five **theorem kinds** based on what they prove:

- `computationalProperty`: Proves algebraic laws about computational definitions (e.g., commutativity of `Nat.add`)
- `mathematicalProperty`: Proves abstract mathematical facts
- `bridgingProperty`: Connects Prop specifications to Bool implementations
- `soundnessProperty`: Proves a user type embeds into a specification
- `completenessProperty`: Proves a specification can be represented by a user type

Bridging properties are the critical "ontological glue"—they validate that implementations correctly reflect specifications. We further classify bridging direction:

- `sound`: `f x = true → P x` (if the algorithm says yes, spec agrees)
- `complete`: `P x → f x = true` (if the spec says yes, algorithm finds it)
- `iff`: Both directions (`f x = true ↔ P x`)

3.4 The Curry-Howard Foundation

Lean's Curry-Howard isomorphism—the correspondence between types and propositions, programs and proofs—makes our four-category ontology practically realizable:

Type Theory	**Lowe's Ontology**
Types in `Prop`	Attributes (non-substantial universals)
Types in `Type`	Kinds/Datatypes (substantial universals)
Terms of `Prop` types	Modes (property instances = proofs)
Terms of `Type` types	Objects (data instances)

This correspondence enables automatic classification: the `Prop`/`Type` universe distinction directly determines mathematical versus computational categories, while dependent types allow properties to vary with their bearers—exactly what Lowe's modes (non-substantial particulars) require. Crucially, Mathlib's formalized mathematics becomes practically useful for software verification: theorems about abstract structures transfer to computational implementations via refinement proofs.

3.5 Bridging Theorem Roles

Bridging theorems serve two purposes [9]: **Validation bridges** (proofs for programs) enable compiler optimizations—e.g., a proof `h : i < arr.size` validates safe array access. **Specification bridges** (programs for proofs) establish functional correctness—e.g., `insert_preserves_balance` verifies an operation against its specification.

Our tool's *validates* relationship captures validation bridges, while *proves* captures specification bridges.

4 Automatic Classification

4.1 Theorem Type Analysis

Given a theorem `theorem T : ∀ (x : α), H₁ → H₂ → ... → C`, we analyze its type to determine what it assumes, proves, and validates. The analysis uses Lean's metaprogramming API to decompose theorem types:

1. **Telescope expansion**: Use `forallTelescope` to bind universal parameters and expose the hypothesis-conclusion structure.
2. **Hypothesis processing**: For each hypothesis H_i that is a `Prop`:
   - Extract head constants (the primary predicate/relation being assumed)
   - Record these as *assumes* relationships
   - Check for Bool-equality patterns (bridging indicators)
3. **Conclusion analysis**: Process the conclusion `C` similarly, recording head constants as *proves* relationships.
4. **Bridging detection**: Identify `f x = true` or `f x = false` patterns indicating Bool-to-Prop connections, recorded as *validates*.

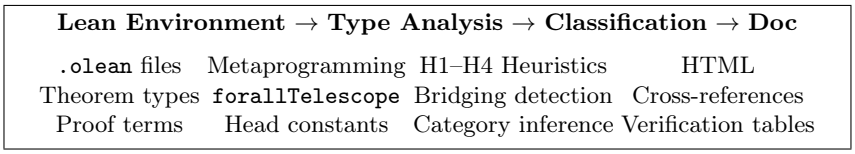

Fig. 1. Documentation generation pipeline. The tool loads compiled Lean modules, analyzes theorem types and proof terms using metaprogramming, applies classification heuristics, and generates navigable documentation.

4.2 Inference Heuristics (H1-H4)

We define four heuristics for extracting semantic relationships from theorem types:

H1 (Prop Hypotheses → Assumes): For each Prop-valued hypothesis, extract the head constant(s) being assumed. Given `H : Chain R a l`, the theorem assumes `Chain`. This captures preconditions and constraints the theorem depends upon.

H2 (Equation Hypotheses → Proves): Equality hypotheses like `H : lookup k m = some v` indicate the theorem proves properties about `lookup`. The equation's head constants are recorded as *proves* relationships.

H3 (Conclusion → Proves): The conclusion's head constants indicate what the theorem establishes. For `TransGen r a b`, the theorem proves properties of `TransGen`.

H4 (Bool Functions → Validates): When a theorem type contains `f x = true` or `f x = false` (or embedded in an `Iff`), the Bool-returning function `f` is being *validated*—shown correct relative to some Prop specification.

4.3 Bridging Detection and Directionality

A theorem is classified as `bridgingProperty` when it connects a Bool-returning computation to a Prop-based specification. The key pattern: if a theorem's type mentions a Bool function in an equation-to-`true/false` context, it bridges computational and mathematical worlds.

For bridging theorems, we infer the correspondence direction: **sound** (`f x = true → P x`) when Bool appears in hypothesis, **complete** (`P x → f x = true`) when in conclusion, **iff** when an `Iff` connects both directions.

5 Implementation

5.1 Architecture Overview

`doc-verification-bridge` is implemented in Lean 4 as a library that extends doc-gen4's documentation pipeline. Figure 1 shows the processing pipeline.

The architecture comprises four main components: **Types.lean** (core type definitions), **Inference.lean** (H1–H4 heuristics and bridging detection), **Classify.lean** (declaration iteration and classification), and **StaticHtml.lean** (HTML generation).

5.2 Core Type Definitions

The ontology is encoded as algebraic data types. `TypeCategory` classifies inductive types and structures:

```
inductive TypeCategory where
  | mathematicalAbstraction  -- Prop-based types (Lowe's "Kinds")
  | computationalDatatype    -- Data-carrying types (Lowe's "Objects")
```

`DefCategory` classifies definitions by return type:

```
inductive DefCategory where
  | mathematicalDefinition   -- Returns Prop (Lowe's "Attributes")
  | computationalOperation   -- Returns non-Prop (Lowe's "Modes")
```

`TheoremKind` captures the five theorem classifications:

```
inductive TheoremKind where
  | computationalProperty   -- Algebraic laws about computations
  | mathematicalProperty    -- Abstract mathematical facts
  | bridgingProperty        -- Bool ↔ Prop connections
  | soundnessProperty       -- UserType → ExternalSpec
  | completenessProperty    -- ExternalSpec → UserType
```

Theorem metadata records the inferred relationships:

```
structure TheoremData where
  kind : Option TheoremKind
  bridgingDirection : Option BridgingDirection
  assumes : Array Name    -- Hypotheses (H1)
  proves : Array Name     -- Conclusion targets (H2, H3)
  validates : Array Name  -- Bool functions validated (H4)
  dependsOn : Array Name  -- Proof dependencies
  axiomDeps : Array Name  -- Axioms used in proof
  hasSorry : Bool         -- Contains incomplete proof
```

5.3 Integration with doc-gen4

The tool integrates with doc-gen4's module loading and analysis infrastructure, generating both standard API documentation and verification coverage reports with cross-links between them.

5.4 Proof Dependency Extraction

Beyond type-based inference, we extract *proof dependencies*—which theorems a proof actually uses. This produces the *dependsOn* relationship, enabling users to trace proof foundations and corroborate the results with blueprint-style dependency graphs.

Proofs as DAGs, Not Trees. Lean's Curry-Howard isomorphism means proofs are programs, and like programs, they benefit from *structural sharing*. A proof term is stored as a directed acyclic graph (DAG) where common subexpressions—especially typeclass instance resolutions and tactic-generated terms—are shared rather than duplicated. This sharing can be extreme: a proof with 1 million unique expression nodes may have billions of "tree nodes" if traversed naively, because the same subexpression (e.g., a `DecidableEq` instance) may be referenced thousands of times.

We employ Lean's built-in `Expr.foldConsts` (from `Lean.Util.FoldConsts`), which uses pointer-based hashing (`PtrSet Expr`) to visit each unique expression node exactly once. This reduces proof dependency extraction from potentially hours to seconds for pathological cases—proofs using tactics like `fun_prop` or extensive `simp` chains can generate proof terms with very high sharing ratios.

Two-Phase Parallel Architecture. The algorithm employs a two-phase approach to maximize parallelism while respecting Lean's threading constraints:

Phase 1 (Sequential, MetaM): Classification requires Lean's `MetaM` monad for type inference and normalization. Since `MetaM` maintains mutable state and is not thread-safe, this phase runs single-threaded. For each declaration, we: (1) classify by analyzing types, (2) apply heuristics H1–H4, and (3) for theorems and lemmas, record a *proof dependency task*—the constant name and proof term expression—for later parallel processing.

Phase 2 (Parallel, IO): Proof term traversal is pure: given an `Expr`, we use `foldConsts` to collect constant references without requiring `MetaM` context. We partition the collected tasks across N worker threads using `IO.asTask`, each traversing proof terms independently. Results are merged after all workers complete.

This separation exploits the observation that type-level analysis requires the environment's elaboration context, while proof-term traversal only requires read access to expression DAGs. For Mathlib4 (commit 721b21c[1]), Phase 1 classified all 279,624 declarations in 23m 29s, Phase 2 analyzed all 223,798 theorems in 1h 2m 59s, and the unified `doc-gen4 + doc-verification-bridge` pipeline ran in 5m 4s serializing 7,427 module pages using a dynamic work queue with 20 worker threads[2].

[1] https://github.com/leanprover-community/mathlib4/commit/721b21cf2bbec8f5709aa776c9f0238a80c8568a

[2] Timing measured on a computer running Red Hat Enterprise Linux 9.6 with an Intel Xeon w7-3465X CPU (56 threads) and 256GB RAM.

5.5 Output Generation

The integration with `doc-gen4` yields a unified pipeline generating a single documentation site combining traditional API reference (via `doc-gen4`) with semantic verification coverage reports. True bidirectional navigation is enabled: API documentation pages include verification badges linking to coverage reports ("what theorems verify this function?"), and coverage report pages link back to API documentation ("what is the API documentation for this definition?").

5.6 Scaling Lean 4 static analysis to large codebases

Analyzing Mathlib4 (490K+ declarations across 7,400+ modules) required engineering a careful balance between Lean's intrinsic threading constraints (e.g., `MetaM`'s single-threaded state) and opportunities for parallelism in pure computations and I/O concurrency. Key techniques include: **streaming serialization** using JSON Lines format to avoid stack overflow when serializing 280K classification entries; **parallel doc generation** writing thousands of html files concurrently; **stack-safe data structure traversal** by converting recursive `NameMap` operations to array-based iteration; and **environment extraction** to run Lake-built executables directly, bypassing I/O buffering issues when Lake acts as an intermediary process. These techniques demonstrate Lean 4's adequacy for high-performance, scalable static analysis of large codebases—an important practical contribution beyond the classification algorithms[3].

6 Empirical Evaluation

6.1 Experimental Setup

We applied `doc-verification-bridge` to 27 Lean 4 projects; however, only 16 were successfully analyzed using compatible versions of Lean between 4.24.0 up to 4.27.0-rc1. Below is a brief summary of projects with large proof developments[4].

Project	Modules	Defs	Theorems	Sorry's
Batteries	164	3,324	1,854	0
Fermat's Last Theorem	175	2,609	1,692	29
PhysLean	377	7,927	5,499	10
Mathlib4	7,427	279,624	223,798	0

[3] For technical details, see: https://github.com/NicolasRouquette/doc-verification-bridge/blob/main/experiments/README.md

[4] See https://nicolasrouquette.github.io/doc-verification-bridge (includes `doc-gen4`)

6.2 Classification Results

The automatic classification is able to categorize all definitions and theorems according to the 4-category ontology; however, careful review is left for future discussion with the Lean community.

Theorem classification is more nuanced. The *bridging* classification identifies relatively few theorems that connect Bool functions to Prop specifications (21 in Batteries, under 1% in Mathlib4), suggesting the heuristics are conservative—they avoid false positives at the cost of missing some valid bridging theorems.

6.3 Threats to Validity

Internal: Heuristics H1–H4 assume conventional theorem formulations; heavily encoded proofs may evade classification. **External**: Results reflect libraries following standard Lean 4 conventions. **Construct**: Coverage captures theorem existence, not proof strength.

7 Related Work

Documentation tools. doc-gen4 [3] generates Lean 4 API documentation but doesn't track theorem-definition relationships. Blueprint [4] requires manual annotations.

Proof dependency analysis. SerAPI (Coq) and CoqGraph extract dependencies; our tool adds semantic classification (bridging, soundness/completeness).

Refinement and verification. Our bridging classification corresponds to data refinement [5, 6]. Why3 [7] and Dafny [8] track specification discharge; we work post-hoc without annotations.

Formal ontology. UFO [10] and BFO [11] use Description Logics, which struggle with Lowe's *modes* [12]. Lean captures both of Lowe's orthogonal axes (Section 3): the `Prop`/`Type` universe distinction corresponds to substantial vs. non-substantial, while the type/term distinction corresponds to universal vs. particular.

8 Discussion: AI-Assisted Proof Engineering

The development of `doc-verification-bridge` was assisted by generative AI (GitHub Copilot with Claude Opus 4.5), yielding observations relevant to formal methods adoption.

The compiler feedback paradox. AI assistance proved more reliable for theorem proving than for documentation formatting. For Lean 4 proofs, the compiler provides immediate feedback on correctness, enabling rapid iteration: incorrect suggestions are caught and refined until the proof compiles. In contrast,

generating HTML tables and CSS styling for MkDocs output required substantially more manual intervention, as there is no analogous "compiler" to verify visual correctness.

Proof strategy exploration. Beyond individual proofs, AI amplifies the exploration of proof *strategies*. Reviewing the AsyncDSLMath library (Appendix A), we identified 14 recurring proof techniques, including: refinement lemmas (soundness/completeness pairs bridging computation and specification), Boolean/Prop equivalence via `List.all_eq_true`, compile-time verification via `native_decide`, Galois connections yielding "theorems for free,"[13] and homomorphism-based theorem lifting through abstraction functions [14]. These techniques were familiar from prior study—Curry-Howard, Galois connections, and refinement are well-established in the literature—but remained largely theoretical knowledge. AI assistance enabled translating this background into working Lean 4 proofs. The development workflow combined human insight (articulating the desired proof strategy) with AI guidance (navigating Lean's APIs and Mathlib's conventions), iterating rapidly until the compiler validated correctness.

Lowering the barrier. This suggests a model for democratizing formal verification: AI transforms proof engineering from "knowing the right approach" to "iteratively searching with compiler-validated feedback." The techniques above represent expert-level knowledge made accessible through AI guidance, potentially lowering the barrier for adopting rigorous verification practices by non-theorem-proving specialists.

9 Conclusion and Future Work

We presented `doc-verification-bridge`, a documentation tool for Lean 4 that automatically extracts verification relationships—what each theorem assumes, proves, validates, and depends on—without requiring annotations. The four-category ontology provides principled classification of declarations into mathematical specifications and computational implementations. Empirical evaluation demonstrates effective classification across multiple libraries.

Future work: Collaborate with the Lean community to apply this tool across the Lean 4 ecosystem and explore potential integration with other tools such as `doc-gen4`. The tool is open source and available[5]; the experiments pipeline was designed for the continuous integration analysis of proof-dense Lean 4 projects.

A AsyncDSLMath: Selected Code

This appendix presents key excerpts from AsyncDSLMath, a formalized architecture description language for asynchronous systems implemented in Lean 4. The code demonstrates a three-layer refinement architecture with bridging theorems.

[5] https://github.com/NicolasRouquette/doc-verification-bridge

A.1 Core Abstractions: MapLike Typeclass

The `MapLike` typeclass abstracts over map implementations, enabling a single specification to apply to both mathematical (`Finmap`) and computational (`ExtTreeMap`) backends.

```
-- MapLike/Basic.lean (excerpt)
@[api_type {category := .mathematicalAbstraction}]
class MapLike (M : Type u) (K : outParam (Type v))
              (V : outParam (Type w)) where
  empty   : M
  lookup? : M → K → Option V
  insert  : K → V → M → M
  erase   : K → M → M
  dom     : M → Set K
  mem_dom_iff : ∀ (m : M) (k : K),
    k ∈ dom m ↔ (lookup? m k).isSome
  ext : ∀ {m₁ m₂ : M},
    (∀ k, lookup? m₁ k = lookup? m₂ k) → m₁ = m₂
```

A.2 Refinement: ComputableMapRefinement

The `ComputableMapRefinement` typeclass captures the relationship between a specification-level map and its efficient implementation.

```
-- MapLike/ComputableMapRefinement.lean (excerpt)
@[api_type {category := .mathematicalAbstraction}]
class ComputableMapRefinement
    (Spec Impl : Type u) (K : outParam (Type v))
    (V : outParam (Type w))
    [MapLike Spec K V] [MapLike Impl K V] where
  -- Abstraction function
  toSpec : Impl → Spec
  -- Computable key enumeration
  keysFinset : Impl → Finset K
  -- Keys match spec domain
  keysFinset_correct : ∀ (m : Impl) (k : K),
    k ∈ keysFinset m ↔ k ∈ MapLike.dom (toSpec m)
  -- Core refinement: lookups agree
  lookup_agrees : ∀ (m : Impl) (k : K),
    MapLike.lookup? (toSpec m) k = MapLike.lookup? m k
```

A.3 Package Registry Specification

The registry is parameterized by any `MapLike` instance, enabling instantiation with either mathematical specifications or efficient implementations.

```
-- Package/Basic.lean (excerpt)
@[api_type {category := .mathematicalAbstraction}]
```

```
structure PackageRegistrySpec (M : Type _)
    [MapLike M PkgName SoftwarePackage] where
  packages : M

-- Direct dependency relation (Prop-valued)
@[api_def {coverage := .complete}]
def depRel (reg : PackageRegistrySpec M) : PkgName → PkgName → Prop :=
  fun p q =>
    match reg.lookup p with
    | none => False
    | some pkg => pkg.dependsOn q

-- Transitive reachability (mathematical specification)
@[api_def {coverage := .complete}]
def depTransitive (reg : PackageRegistrySpec M) : PkgName → PkgName → Prop :=
  Relation.TransGen (depRel reg)
```

A.4 Graph Theory: PathWithLength

We define a Prop-indexed path structure that bridges to Mathlib's `TransGen` and `Quiver.Path`, enabling graph-theoretic reasoning.

```
-- Abstractions/GraphTheory.lean (full definitions)
@[api_type {category := .mathematicalAbstraction}]
inductive PathWithLength {α : Type*} (r : α → α → Prop)
    : α → α → Nat → Prop
  | single {a b} : r a b → PathWithLength r a b 1
  | cons {a b c n} : r a b → PathWithLength r b c n →
                     PathWithLength r a c (n + 1)

-- Soundness: PathWithLength implies TransGen
@[api_theorem {theoremKind := .soundnessProperty}]
theorem PathWithLength_soundness {a b : α} {n : Nat}
    (h : PathWithLength r a b n) : Relation.TransGen r a b := by
  induction h with
  | single hab => exact Relation.TransGen.single hab
  | cons hab _ ih =>
    exact Relation.TransGen.trans (Relation.TransGen.single hab) ih

-- Completeness: TransGen yields PathWithLength
@[api_theorem {theoremKind := .completenessProperty}]
theorem PathWithLength_completeness {a b : α}
    (h : Relation.TransGen r a b) : ∃ n, PathWithLength r a b n := by
  induction h with
  | single hab => exact ⟨1, PathWithLength.single hab⟩
  | tail _ hbc ih =>
    obtain ⟨n, pwl⟩ := ih
    exact ⟨n + 1, pwl.PathWithLength_append hbc⟩
```

A.5 Graph Theory Axioms

We axiomatize standard results in graph theory on simple paths. These could be proven from first principles, but are well-known results. The 3 axioms below are temporary placeholders for ongoing work by Matteo Cipollina on Markov Chain Monte-Carlo in Lean 4[6].

```
-- Graph Theory Axioms
axiom quiver_simple_path_bound (V : Type*)
    [DecidableEq V] [Quiver V] [Fintype V]
    {a b : V} (p : Path a b) (h_simple : p.vertices.Nodup) :
    p.length < Fintype.card V

axiom quiver_exists_simple_subpath (V : Type*)
    [DecidableEq V] [Quiver V]
    {a b : V} (p : Path a b) :
    ∃ (q : Path a b), q.vertices.Nodup ∧ q.length ≤ p.length

axiom quiver_shortest_path_bound (V : Type*)
    [DecidableEq V] [Quiver V] [Fintype V]
    {a b : V} (p : Path a b) (h : p.length > 0) :
    ∃ (q : Path a b), q.length > 0 ∧ q.length < Fintype.card V
```

A.6 Main Graph Theory Result

The key theorem: any path has a simple subpath bounded by vertex count.

```
-- Bounded Path Length Theorem
@[api_theorem {theoremKind := .mathematicalProperty}]
theorem PathWithLength_bounded {α : Type*} [DecidableEq α] [Fintype α]
    (r : α → α → Prop) {a b : α} {n : Nat}
    (h : PathWithLength r a b n) :
    ∃ m, PathWithLength r a b m ∧ m < Fintype.card α := by
  -- 1. Convert PathWithLength (Prop) to PathWithLengthData (Type)
  let p_data := h.toData
  -- 2. Convert to Quiver.Path
  let p_quiver := p_data.toQuiverPath
  -- 3. Apply graph theory axiom
  have h_bounded := quiver_shortest_path_bound _ p_quiver (by omega)
  obtain ⟨q, _, h_q_bound⟩ := h_bounded
  -- 4. Convert back to PathWithLength
  let q_data := PathWithLengthData.ofQuiverPath q (by omega)
  exact ⟨q.length, q_data.toProp, h_q_bound⟩
```

[6] See Lean 4 ZulipChat discussion: https://leanprover.zulipchat.com/#narrow/channel/287929-mathlib4/topic/Proving.20Graph-Theoretic.20Properties.20for.20Quiver.20Paths

A.7 Bounded Path Search Algorithm

The implementation layer provides a decidable path search with fuel (iteration bound).

```
-- Package/Basic.lean: Implementation layer (excerpt)
namespace PackageRegistryImpl

-- Bounded DFS for path detection
def hasPathWithFuelAux (reg : PackageRegistrySpec Impl)
    (start target : PkgName) (fuel : Nat) (tookStep : Bool) : Bool :=
  match fuel with
  | 0 => tookStep && (start == target)
  | fuel' + 1 =>
    if tookStep && start == target then true
    else match lookup reg start with
      | none => false
      | some pkg => pkg.dependencies.any fun dep =>
          hasPathWithFuelAux reg dep.name target fuel' true

def hasPathWithFuel (reg : PackageRegistrySpec Impl)
    (start target : PkgName) (fuel : Nat) : Bool :=
  hasPathWithFuelAux reg start target fuel false

-- Registry size as fuel bound
def size [ComputableMapRefinement Spec Impl PkgName SoftwarePackage]
    (reg : PackageRegistrySpec Impl) : Nat :=
  (ComputableMapRefinement.keysFinset reg.packages).card

end PackageRegistryImpl
```

A.8 Bridging Theorems: Soundness and Completeness

These theorems establish that the bounded search correctly implements the transitive reachability specification.

```
-- Package/Computable.lean: Core bridging theorems

-- Soundness: if search returns false, no path exists
@[api_theorem {theoremKind := .bridgingProperty,
               bridgingDirection := .sound,
               validates := #[`hasPathWithFuel]}]
theorem hasPathWithFuel_sound (h_wf : reg.wellFormed)
    (start target : PkgName) :
    hasPathWithFuel reg start target (size reg) = false →
    ¬ depTransitive reg start target := by
  intro h_false h_path
  -- Contrapositive: path exists implies search succeeds
  have h_true := hasPathWithFuel_mem_iff_registry ⟨reg, h_wf⟩
                   start target h_path
```

```
  rw [h_true] at h_false
  contradiction

-- Completeness: if path exists, search finds it
@[api_theorem {theoremKind := .bridgingProperty,
               bridgingDirection := .complete,
               validates := #[`hasPathWithFuel]}]
theorem hasPathWithFuel_complete (h_wf : reg.wellFormed)
    (start target : PkgName)
    (h_path : depTransitive reg start target) :
    hasPathWithFuel reg start target (size reg) = true := by
  -- 1. Get bounded PathWithLength
  obtain ⟨n, h_pwl, h_n_le⟩ := exists_simple_path_bound_wellFormed
                                ⟨reg, h_wf⟩ h_path
  -- 2. DFS detects path at exact length
  have h_detect := hasPathWithFuel_detects_path_len reg h_pwl
  -- 3. Monotonicity: more fuel preserves detection
  exact hasPathWithFuel_correct_of_le reg start target h_n_le h_detect

-- Decidable dependency relation (iff bridging)
@[api_theorem {theoremKind := .bridgingProperty,
               bridgingDirection := .iff,
               validates := #[`depRel]}]
theorem depRel_decidable_correct (p q : PkgName) :
    decide (PackageRegistryImpl.depRel reg p q) = true ↔
    PackageRegistrySpec.depRel reg p q := by
  unfold PackageRegistryImpl.depRel
  simp [decide_eq_true_eq]
```

A.9 Classification Summary

The `doc-verification-bridge` tool automatically produces the following classification for the key AsyncDSLMath declarations:

Declaration	Category/Kind	Role
`MapLike`	mathematicalAbstraction	Spec interface
`ComputableMapRefinement`	mathematicalAbstraction	Refinement relation
`PackageRegistrySpec`	mathematicalAbstraction	Registry spec
`depRel`	mathematicalDefinition	Direct dependency (Prop)
`depTransitive`	mathematicalDefinition	Transitive closure (Prop)
`hasPathWithFuel`	computationalOperation	Bounded DFS (Bool)
`size`	computationalOperation	Registry size
`PathWithLength_soundness`	soundnessProperty	Path → TransGen
`PathWithLength_completeness`	completenessProperty	TransGen → Path
`hasPathWithFuel_sound`	bridgingProperty (sound)	Search validates spec
`hasPathWithFuel_complete`	bridgingProperty (complete)	Spec validates search
`depRel_decidable_correct`	bridgingProperty (iff)	Decidability

The dependency analysis reveals the proof structure: `hasPathWithFuel_complete` depends on `PathWithLength_completeness` (graph theory), `exists_simple_path_bound` (vertex bound), and `hasPathWithFuel_correct_of_le` (monotonicity). This chain shows how Mathlib's abstract graph theory connects to executable algorithms.

Acknowledgments. The research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (80NM0018D0004). Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology. AI assistance (GitHub Copilot) was used as discussed in Section 8.

References

1. Lowe, E.J.: The Four-Category Ontology: A Metaphysical Foundation for Natural Science. Oxford University Press (2006)
2. Coquand, T., Huet, G.: The Calculus of Constructions. Information and Computation **76**(2-3), 95–120 (1988)
3. doc-gen4: Documentation generator for Lean 4. https://github.com/leanprover/doc-gen4
4. Massot, P.: Lean Blueprint. https://github.com/PatrickMassot/leanblueprint
5. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer (1998)
6. Bjørner, D., Jones, C.B. (eds.): The Vienna Development Method: The Meta-Language. LNCS, vol. 61. Springer (1978)
7. Filliâtre, J.C., Paskevich, A.: Why3 — Where Programs Meet Provers. In: ESOP 2013, LNCS, vol. 7792, pp. 125–128. Springer (2013)
8. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: LPAR 2010, LNCS, vol. 6355, pp. 348–370. Springer (2010)
9. Himmel, M.: Proofs for Programs, Programs for Proofs. BOB 2026, Berlin. https://bobkonf.de/2026/himmel.html
10. Guizzardi, G.: Ontological Foundations for Structural Conceptual Models. PhD thesis, University of Twente (2005)
11. Arp, R., Smith, B., Spear, A.D.: Building Ontologies with Basic Formal Ontology. MIT Press (2015)
12. Dapoigny, R., Barlatier, P.: Modeling Ontological Structures with Type Classes in Coq. In: FOIS 2010, Frontiers in Artificial Intelligence, vol. 209, pp. 135–148. IOS Press (2010)
13. Backhouse, K., Backhouse, R.: Safety of abstract interpretations for free, via logical relations and Galois connections. In: Science of Computer Programming, Volume 51, Issues 1–2, 2004
14. Hoare, C.A.R.: Proof of Correctness of Data Representations. Acta Informatica **1**(4), 271–281 (1972)