From Lean Proofs to Axiom-Free Executables: A Specification for Verified Compilation, Diverse-Kernel Validation, and Higher Semantics

Anonymous

27 October 2026
Prepared for the Meta Journal of Agentic Meta Algorithms

Abstract

We specify a verifiable pipeline from constructive Lean developments to executable programs with no declared axioms in their audited dependency closures. A proof of existence in Lean's proof-irrelevant proposition universe is distinguished from a witness-producing construction in its data universe. We separate logical axiom freedom, runtime computability, and correctness of compilation. The initial implementation target is a precisely defined arithmetic language with proof annotations, lexical binding, branching, and bounded iteration, together with a structured stack machine. We give erasure, totality, stack-preservation, and semantic-preservation arguments, and an interface for proof-producing translation from Lean declarations. Dependent refinements are interpreted through sections and predicate fibrations; realizability supplies an effectiveness condition; higher-topos semantics explains the difference between local existence and globally chosen data. Homotopical semantics is external to Lean's proof-irrelevant equality and is not identified with runtime erasure. A release protocol combines empty dependency audits, theorem-statement binding, independently implemented kernels, and reproducible artifacts. The result is a specification and verification plan, not a claim that a general Lean extraction compiler or an end-to-end verified native toolchain has already been implemented.

Keywords: Lean; constructive extraction; verified compilers; proof erasure; realizability; higher toposes; homotopy semantics; proof-checker diversity.

1 Purpose, scope, and status

Agent-generated proofs can be useful inputs to software development only if the intended theorem, the computational witness, the translation, and the validation evidence remain connected. A valid theorem about an unintended program is insufficient. A compiled program accompanied by an unrelated valid theorem is likewise insufficient. Our object of study is therefore a *bound artifact*: a specified input/output relation, a witness-producing program, a checked proof of its relation to the specification, an explicitly defined compilation, and a record identifying exactly what independent checkers accepted.

The intended sequence is: specification; verified Lean implementation; validation by different kernels; then a release with an explicit assurance boundary. This manuscript supplies the first stage and a small executable reference model. Its mathematical arguments are presented for subsequent formalization. Unless the accompanying evidence explicitly says otherwise, they are not reported as mechanically checked Lean proofs.

The main engineering contribution is the combination of a restrictive acceptance policy, a finite initial language, explicit compiler rules, and release predicates. The underlying distinctions

between proofs and programs, realizability, propositional truncation, and relative soundness are established ideas. We use them to state obligations rather than presenting them as newly discovered theorems. The specification deliberately rejects unsupported input. It does not promise automatic extraction from arbitrary Lean mathematics, arbitrary axiomatic existence proofs, arbitrary dependent recursors, or arbitrary effectful programs.

1.1 Three meanings that must not be conflated

For a source declaration f and a claimed correctness proof p, distinguish:

1. **Logical axiom freedom:** the transitive declaration dependencies of f, its type, p, and the type of p contain no axiom declarations.
2. **Computational admissibility:** retained operations have operational implementations, supported representations, and a justified termination argument.
3. **Verified compilation:** the actual translation preserves the specified source behavior under stated source, target, and representation semantics.

None alone implies the other two. A noncomputable choice can occur only inside an erased proof of a computable function. Conversely, an axiom-free definition can use a recursor unsupported by a particular compiler. A source proof checked by a kernel does not by itself establish the correctness of the compiler that emits native code. Lean's documentation explicitly separates these matters [2, 3, 4, 5].

2 What is being extracted?

2.1 Existence propositions and witness-producing types

Let $A : \mathsf{Type}$, $B : A \to \mathsf{Type}$, and $R : \prod_{x:A} B(x) \to \mathsf{Prop}$. The following are different interfaces:

$$H : \forall x : A,\ \exists y : B(x),\ R(x, y), \tag{1}$$

$$F : (x : A) \to \{y : B(x) \mid R(x, y)\}. \tag{2}$$

The first lives in Prop. The second returns data with a proof field. In Lean, an ordinary existential proof cannot generally be eliminated into an arbitrary data type. By contrast, $f(x) := (F(x)).\mathsf{val}$ is a program, and $(F(x)).\mathsf{property}$ certifies it. For a type-valued specification, use an actual dependent sum $\sum_{y:B(x)} S(x, y)$, expressed with `Sigma`; for a proposition-valued property, use `Subtype`. Lean's ordinary `Sigma` does not accept an arbitrary `Prop`-valued second family as though `Prop` and `Type` were interchangeable.

Proposition 2.1 (Projection of a verified construction)**.** *Given a definable F of type (2), projection defines $f : \prod_{x:A} B(x)$ with a proof $\forall x, R(x, f(x))$, without adding a choice axiom.*

Proof. Set $f(x) = (F(x)).\mathsf{val}$ and use the second projection of $F(x)$ as the certificate. Both projections are primitive structure eliminations. No witness is recovered from an erased existential proof. □

A particular constructive existential proof may expose a witness to a metaprogram that inspects its syntax. That possibility is not a uniform Lean function from arbitrary existential proofs to data, and is not a license to assume a general extraction theorem. A practical alternative is to write the algorithm first and prove it satisfies the relation. Decidable finite search is another alternative: the decision procedure and search bound are computational inputs and must themselves pass the audit.

2.2 A limit on general extraction

Proposition 2.2 (Classical uniqueness does not ensure computability)**.** *There is a classically total, single-valued relation $R \subseteq \mathbb{N} \times \{0,1\}$ with no total computable selector.*

Proof. Fix an effective enumeration of machines and let $\mathsf{Halt}(e)$ mean that machine e halts on its own code. Put

$$R(e,b) \iff (b = 1 \wedge \mathsf{Halt}(e)) \vee (b = 0 \wedge \neg\mathsf{Halt}(e)).$$

Classical excluded middle proves existence of exactly one such b. A total computable selector decides the diagonal halting problem, contradicting its undecidability. Thus even uniqueness does not repair the missing algorithm. □

This is a metatheoretic obstruction, not an accepted axiom-free source program. Any compiler advertised as extracting algorithms from *all* classical existence theorems is already inconsistent with this example.

3 Axiom and computational dependency policies

3.1 Logical closure

Let D be a finite declaration environment. A node records the exact name, universe parameters, declaration kind, type, body when present, primitive status, and relevant attributes. Define $\mathsf{refs}(d)$ to include every constant in its type and body and all structural dependencies of inductive blocks, constructors, and recursors. Opaque theorem bodies must be obtained from the authenticated export; opacity is not permission to omit dependencies.

For roots S, define $\mathsf{Reach}_D(S)$ as the least subset containing S and closed under refs. Define

$$\mathsf{Ax}_D(S) = \{a \in \mathsf{Reach}_D(S) \mid a \text{ is an axiom declaration}\}.$$

The **strict profile** requires $\mathsf{Ax}_D(S) = \varnothing$. Roots include the program, its specification, its correctness proof, the translation certificates, and the compiler-correctness theorems used by the release. Open proof parameters must be discharged, or listed as explicit preconditions in the public theorem. A declaration $P \to Q$ with no axioms is not a proof of Q.

This rejects `Classical.choice`, `propext`, `Quot.sound`, `sorryAx`, user axioms, and native-evaluation axioms whenever reachable. Even though some of these are compatible with proof-erasing compilation, the strict policy is intentionally stronger. Proof irrelevance and the kernel's typing and reduction rules are part of the foundational theory; an empty axiom list is not a statement that there are no foundational assumptions.

Proposition 3.1 (Finite audit)**.** *On a finite, fully disclosed environment, worklist traversal computes $\mathsf{Reach}_D(S)$ and its axiom subset, and terminates.*

Proof. Maintain a worklist and a visited set. Remove one unvisited node, add it to visited, and enqueue its dependencies. Each node enters visited at most once, so at most $|D|$ expansions occur. Induction on the traversal shows that visited nodes are reachable. Induction on dependency-path length shows that every reachable node is eventually visited. Filtering by declaration kind yields exactly the stated axiom set. With indexed adjacency lists the graph work is $O(|V| + |E|)$, excluding parsing and proof checking. □

Missing bodies, duplicate declaration identifiers, unresolved names, malformed universe data, and ambiguous primitive encodings are errors. The graph is computed from typed artifacts, not from searching source text for the word `axiom`. The final checker must validate the graph or independently reconstruct it; an untrusted exporter can otherwise omit an edge.

3.2 Runtime closure and conservative rejection

Runtime dependencies concern the validated, erased representation. A simple graph difference between data nodes and proof nodes is not sufficient: proof arguments can justify casts, termination, and dependent elimination. Deleting them requires a preservation theorem. The strict first implementation rejects:

- unsupported axioms, quotient operations, and data extraction through choice;
- `unsafe`, unrestricted `partial`, unchecked foreign calls, runtime replacement attributes, and unsupported opaque computational constants;
- arbitrary type-directed runtime computation or equality transport without a representation theorem;
- unsupported recursive equations, effectful primitives, and any unresolved computation budget.

Trusted implementations of arithmetic or memory operations, if used, must be named in a primitive ledger with a refinement obligation. Rejecting all foreign implementations is a property of the verified object language, not a claim that Lean's host runtime contains no C code. Likewise `noncomputable` is a useful rejection flag, not a decision procedure for mathematical noncomputability. The initial frontend may conservatively reject such a declaration even when a separate executable implementation exists.

The distinction between kernel reduction and compiled evaluation also matters. `#reduce` and `#eval` have different mechanisms. Proof-based transports can obstruct kernel normalization while being erased by compilation [6]. Source normal-form equality must not be substituted silently for runtime equivalence.

4 The initial source language

To make the implementation finite and inspectable, version 1 supports arithmetic data over unbounded natural numbers. Booleans can be represented by tested numerical tags; richer data and dependent eliminators are later extensions, each with its own proof. The source is a deeply embedded language inside Lean, not all of `Lean.Expr`.

Let E_n be well-scoped expressions with n natural-number variables. Environments are vectors $\rho \in \mathbb{N}^n$, with index zero referring to the newest binder. The grammar is

$$\begin{aligned} e \in E_n \;::=\; & \underline{k} \mid \mathsf{var}(i) \mid \mathsf{add}(e,e) \mid \mathsf{let}(e,b) \\ & \mid \mathsf{if0}(c,t,f) \mid \mathsf{loop}(m,z,b) \mid \mathsf{ann}(P,h,e), \end{aligned}$$

where $k \in \mathbb{N}$, $i : \mathsf{Fin}(n)$, $b \in E_{n+1}$ for let, and $b \in E_{n+2}$ for loop. For ann, $P : \mathsf{Prop}$ and $h : P$ are static annotations. An annotation does not by itself assert a property of the enclosed expression. Semantic properties are supplied by the separate certificate below; this prevents an unrelated true proposition from being mistaken for program correctness.

Write $a :: \rho$ for environment extension and define $I(F, z, 0) = z$ and

$$I(F, z, k+1) = F(k, I(F, z, k)).$$

The denotation is fixed by the following equations:

$$\begin{aligned}
[\![\underline{k}]\!]\rho &= k, & [\![\mathsf{var}(i)]\!]\rho &= \rho_i,\\
[\![\mathsf{add}(a,b)]\!]\rho &= [\![a]\!]\rho + [\![b]\!]\rho,\\
[\![\mathsf{let}(v,b)]\!]\rho &= [\![b]\!]([\![v]\!]\rho :: \rho),\\
[\![\mathsf{if0}(c,t,f)]\!]\rho &= \begin{cases} [\![t]\!]\rho & [\![c]\!]\rho = 0,\\ [\![f]\!]\rho & \text{otherwise,}\end{cases}\\
[\![\mathsf{loop}(m,z,b)]\!]\rho &= I((i,a) \mapsto [\![b]\!](a :: i :: \rho), [\![z]\!]\rho, [\![m]\!]\rho),\\
[\![\mathsf{ann}(P,h,e)]\!]\rho &= [\![e]\!]\rho.
\end{aligned}$$

The body of a loop sees accumulator at index zero and iteration index at index one. The bound and seed are evaluated once. The environment outside the loop is unchanged. These choices are part of the specification and must be covered by tests.

Definition 4.1 (Certified source)**.** *For $P : \mathbb{N}^n \to \mathsf{Prop}$ and $R : \mathbb{N}^n \to \mathbb{N} \to \mathsf{Prop}$, a certified source is a pair (e, p) with*

$$e : E_n, \qquad p : \forall \rho,\ P(\rho) \to R(\rho, [\![e]\!]\rho).$$

Both e and p must satisfy the declared dependency policy. A callable interface must establish P or check a computable predicate whose equivalence to P is proved.

Proposition 4.2 (Totality of source evaluation)**.** *Every $e : E_n$ has a total denotation $\mathbb{N}^n \to \mathbb{N}$.*

Proof. Proceed by structural induction on e. Variables are bounded by their indices; the arithmetic, binding, and branch clauses compose total functions. In the loop case, the induction hypothesis gives a total body function. A separate induction on the evaluated natural bound proves totality of I. The annotation case reduces to its subexpression. This proof establishes mathematical termination, not a uniform feasible time or memory bound. □

4.1 Erasure

The data language E_n^0 has the same grammar without ann. Erasure is homomorphic on all data constructors and

$$\mathsf{erase}(\mathsf{ann}(P,h,e)) = \mathsf{erase}(e).$$

Theorem 4.3 (Erasure preservation)**.** *For $e : E_n$ and $\rho \in \mathbb{N}^n$, $[\![\mathsf{erase}(e)]\!]_0\rho = [\![e]\!]\rho$.*

Proof. Structural induction handles literals, variables, addition, binding, and branching by their displayed equations. In the binding case, apply the body induction hypothesis at the extended environment. For a loop, bound and seed agree by induction. If body evaluations agree for all extended environments, induction on k gives equality of the two I computations: the zero case is the common seed; the successor case uses the equal previous accumulators and the body induction hypothesis. An annotation is erased to its body and both source denotations are the body's denotation. □

This is a theorem about the specified syntax. It is not a proof that an arbitrary transformation of Lean's dependent core preserves semantics.

5 Structured stack target and compiler

Stacks contain natural numbers with the top at the left. A code block C is built from

$$\mathsf{PUSH}(k),\ \mathsf{LOAD}(i),\ \mathsf{ADD},\ C;C,\ \mathsf{ENTER}(C),\ \mathsf{BR}(C,C),\ \mathsf{LOOP}(C).$$

The typing judgment $n \vdash C : h \Rightarrow k$ states that a block under an environment of length n transforms a stack of length h into one of length k. The primitive rules are:

Instruction or rule	Stack effect and premise
$\mathsf{PUSH}(a)$, $\mathsf{LOAD}(i)$	$h \Rightarrow h+1$; for load, $i < n$
ADD	$h+2 \Rightarrow h+1$
$C; D$	$h \Rightarrow l$ if $C : h \Rightarrow k$, $D : k \Rightarrow l$
$\mathsf{ENTER}(B)$	$h+1 \Rightarrow h+1$ if $n+1 \vdash B : h \Rightarrow h+1$
$\mathsf{BR}(T,F)$	$h+1 \Rightarrow k$ if $n \vdash T, F : h \Rightarrow k$
$\mathsf{LOOP}(B)$	$h+2 \Rightarrow h+1$ if $n+2 \vdash B : h \Rightarrow h+1$

The natural h is universally quantified in the expression compiler's result. Block verification can infer effects relative to a given starting height, using structural recursion. Version 1 emits structured blocks rather than raw jump offsets, avoiding a premature trusted assembler.

5.1 Machine equations

The raw interpreter returns either a stack or a tagged error. On well-typed blocks its equations are:

$$\begin{aligned}
\mathsf{run}(\mathsf{PUSH}(a), \rho, s) &= a :: s,\\
\mathsf{run}(\mathsf{LOAD}(i), \rho, s) &= \rho_i :: s,\\
\mathsf{run}(\mathsf{ADD}, \rho, b :: a :: s) &= (a+b) :: s,\\
\mathsf{run}(C; D, \rho, s) &= \mathsf{run}(D, \rho, \mathsf{run}(C, \rho, s)),\\
\mathsf{run}(\mathsf{ENTER}(B), \rho, a :: s) &= \mathsf{run}(B, a :: \rho, s),\\
\mathsf{run}(\mathsf{BR}(T,F), \rho, c :: s) &= \begin{cases} \mathsf{run}(T, \rho, s) & c = 0,\\ \mathsf{run}(F, \rho, s) & c \neq 0. \end{cases}
\end{aligned}$$

In the sequential equation, errors propagate and are never coerced to numbers. To evaluate $\mathsf{LOOP}(B)$ on $z :: m :: s$, put $a_0 = z$. For $0 \le i < m$, run B under $a_i :: i :: \rho$ with stack s, obtain $a_{i+1} :: s_{i+1}$, and use s_{i+1} as the next iteration's base stack. Return the final accumulator above the final base stack. For compiler-generated bodies, the stronger frame theorem below gives $s_{i+1} = s$ throughout. An implementation must not silently reset a modified base stack for arbitrary well-typed input; stack height alone does not imply preservation of its contents.

Theorem 5.1 (Typed machine totality)**.** *If $n \vdash C : h \Rightarrow k$, $|\rho| = n$, and $|s| = h$, then the machine returns a stack of length k, without a lookup or stack-shape error.*

Proof. Induct on the typing derivation. Load bounds and addition's two operands follow from their premises. Composition uses the intermediate height. Enter extends the environment and removes the argument. Both branch arms have the required common effect. For loop, the body typing gives one returned accumulator and a base stack of the original height; induction on the finite bound supplies every iteration. Raw decoding and resource exhaustion are outside this ideal totality theorem and are addressed separately. □

5.2 Translation rules

For $e : E_n^0$, define $\mathsf{compile}_n(e)$ recursively:

$$\begin{aligned}
\mathsf{compile}(\underline{k}) &= \mathsf{PUSH}(k), & \mathsf{compile}(\mathsf{var}(i)) &= \mathsf{LOAD}(i),\\
\mathsf{compile}(\mathsf{add}(a,b)) &= \mathsf{compile}(a);\mathsf{compile}(b);\mathsf{ADD},\\
\mathsf{compile}(\mathsf{let}(v,b)) &= \mathsf{compile}(v);\mathsf{ENTER}(\mathsf{compile}(b)),\\
\mathsf{compile}(\mathsf{if0}(c,t,f)) &= \mathsf{compile}(c);\mathsf{BR}(\mathsf{compile}(t),\mathsf{compile}(f)),\\
\mathsf{compile}(\mathsf{loop}(m,z,b)) &= \mathsf{compile}(m);\mathsf{compile}(z);\mathsf{LOOP}(\mathsf{compile}(b)).
\end{aligned}$$

These rules also determine evaluation order. The compiler performs no search for a proof and does not invoke an oracle.

Theorem 5.2 (Stack effect and semantic preservation)**.** *For every $e : E_n^0$ and h, $n \vdash \mathsf{compile}(e) : h \Rightarrow h+1$. Moreover, for every $\rho \in \mathbb{N}^n$ and every stack s,*

$$\mathsf{run}(\mathsf{compile}(e), \rho, s) = [\![e]\!]_0\rho :: s. \tag{3}$$

Proof. The stack effects follow by structural induction and the typing rules. For (3), literals and variables are immediate. For addition, the first compiled operand puts $a = [\![e_1]\!]\rho$ on s; the second puts $b = [\![e_2]\!]\rho$ on $a :: s$; addition returns $(a+b) :: s$. For binding, the value is pushed and enter invokes the compiled body at $a :: \rho$ with the original s, to which its induction hypothesis applies. For a branch, the condition is pushed and removed before the selected arm runs; its induction hypothesis applies with the original stack. For a loop, the bound and seed become $z :: m :: s$. At iteration i, the body induction hypothesis gives exactly $[\![b]\!](a_i :: i :: \rho) :: s$. Thus the base stack remains s, and induction on i identifies the accumulator sequence with I. After m iterations the result is the source denotation above s. This proves the stronger frame property, not just equality of final stack lengths. □

Corollary 5.3 (Certified execution)**.** *Let (e, p) be certified and assume $P(\rho)$. Running the erased, compiled expression on the empty stack returns $[v]$ with $R(\rho, v)$.*

Proof. Erasure preservation and (3) identify v with $[\![e]\!]\rho$. Substitute this equality in $p(\rho)$ applied to the precondition proof. □

All recursions here are structural or bounded by an explicitly computed natural. The compiler runs in time linear in expression-tree size, counting output constructors and ignoring arbitrary-precision integer copying and shared-term expansion. This is a compiler construction bound, not a bound on execution of nested source loops.

6 Connecting actual Lean declarations to the core

The first implementation milestone accepts the embedded syntax directly. To accept an ordinary Lean declaration $f : \mathbb{N}^n \to \mathbb{N}$, an untrusted frontend may propose $e : E_n$ and a proof term

$$q : \forall \rho,\ f(\rho) = [\![e]\!]\rho.$$

The kernel checks q, and the strict audit traverses q together with f, e, and their types. This *translation-validation* boundary permits an incomplete, heuristic, or agent-written frontend without trusting its correctness. It does not permit that frontend to choose a different specification silently.

The original source declaration and the expected type of q are fixed by the release request. A checker must construct or compare the expected theorem statement from those fixed inputs. Accepting a theorem of any type supplied by the candidate is insufficient. The same rule applies to the final correctness theorem. A digest identifies bytes; semantic binding requires the theorem's term references and types to match the intended declarations.

Theorem 6.1 (Frontend composition)**.** *Suppose q is as above and $p : \forall \rho, P(\rho) \to R(\rho, f(\rho))$. If $P(\rho)$, running $\mathsf{compile}(\mathsf{erase}(e))$ yields $[v]$ satisfying $R(\rho, v)$.*

Proof. The compiler theorem gives $v = [\![e]\!]\rho$. The frontend certificate gives $f(\rho) = [\![e]\!]\rho$. Equality substitution transports $p(\rho)$ to $R(\rho, v)$. □

Version 1 may recognize literal arithmetic, fixed input vectors, lexical binding, numerical tests, and bounded folds. If a recursive Lean function requires unbounded well-founded recursion, arbitrary inductive families, higher-order closures, quotient elimination, or data-dependent layouts, the frontend returns `unsupported`. Later extensions require a new semantic clause, representation relation, translation rule, and preservation proof. Kernel checking of the source does not fill in these missing compiler obligations.

6.1 Native executables and the backend gap

The verified endpoint above is a structured bytecode program interpreted by a specified machine. A Lean implementation of that interpreter can be compiled with Lean's existing toolchain, but its native executable additionally depends on that toolchain, foreign arithmetic, allocation, runtime, operating system, and hardware. Cross-checking the interpreter's proof does not remove these dependencies.

For a further backend B and representation relation $\approx$, require a theorem of the form

$$s \approx \hat{s} \ \wedge \ \mathsf{run}(C, \rho, s) = s' \implies \mathsf{exec}_{\mathrm{target}}(B(C), \hat{\rho}, \hat{s}) \Downarrow \hat{s}' \ \wedge \ s' \approx \hat{s}'.$$

This statement needs explicit environment representation, a semantics for the target instructions, and adequate resource hypotheses. With bounded memory, resource exhaustion must be an explicit outcome. For fuel-bounded execution, prove that *if* a result is returned it is correct, and separately prove the existence of sufficient fuel; these are different obligations. Do not replace unbounded naturals by machine integers without an overflow theorem or an explicit modular specification.

7 Categorical and topos semantics

7.1 Contexts, substitutions, and refinements

Interpret a context of length n as $\mathbb{N}^n$. A substitution $n \to m$ is an m-tuple of expressions in E_n, interpreted componentwise as a function $\mathbb{N}^n \to \mathbb{N}^m$. Renaming and simultaneous substitution are capture-avoiding. Their fundamental equation is

$$[\![e[\sigma]]\!](\rho) = [\![e]\!]([\![\sigma]\!](\rho)).$$

It is proved by induction on e, lifting substitutions under binders and both loop-bound variables. The loop case also uses induction on the numerical bound, as in erasure preservation. This lemma is a required formalization obligation before claiming an implemented syntactic category.

Quotient these substitutions externally by pointwise denotational equality to obtain a category $\mathcal{C}$. Identities are variable tuples and composition is substitution. The substitution lemma establishes

associativity and units semantically. In the strict Lean implementation, use *setoid-enriched homs* and prove laws pointwise; do not introduce `Quot.sound` or `funext` merely to package extensional equality.

Above a context n, consider predicates $P : \mathbb{N}^n \to \mathsf{Prop}$. Reindexing along $f : n \to m$ is $f^*Q = Q \circ [\![f]\!]$. This is the standard fibred account of predicates and dependent types [13]. A refinement morphism $(n, P) \to (m, Q)$ consists of a substitution f and a proof

$$\forall \rho,\ P(\rho) \to Q([\![f]\!](\rho)).$$

Forgetting this certificate yields the computational map. Composition uses implication and the substitution lemma. This predicate fibration captures why specifications can be erased while the witnessed data map remains. It is a statement about a typed semantic projection, not permission to discard every node whose spelling looks like a proof.

7.2 Witnesses are sections

In a locally cartesian closed category with appropriate interpretations, a dependent family $B(x)$ over A is a map $p : B \to A$. A dependent function is a section $s : A \to B$ with $p \circ s = 1_A$. A subtype relation is a subobject of the total space; a verified construction is a section landing in that subobject. Dependent sums compose display maps and dependent products are right adjoints to pullback.

Proposition 7.1 (Existence and sections)**.** *In a topos, internally asserting $\forall x : A, \exists y : B_x$ expresses that $p : B \to A$ is an epimorphism. A witness-producing family is a chosen section. An epimorphism need not have a section.*

Proof. The image subobject of p interprets the existential predicate. Its equality with A is exactly epimorphicity. The semantics of dependent functions identifies a global witness with a section. For a counterexample, take the sheaf topos on S^1 and the sheaf of local sections of the covering $z \mapsto z^2$. Local square roots exist, so the map of this sheaf to the terminal sheaf is epic. A global section would give a continuous $s : S^1 \to S^1$ with $s(z)^2 = z$. On fundamental groups this would require $2k = 1$ for an integer k, a contradiction. □

The counterexample concerns local versus global existence. It is not a claim that every epimorphism in ordinary classical sets fails to split, or that every global section is computable. Classical sets can possess noncomputable sections, so a further effectiveness condition is necessary.

7.3 Realizability and the effectiveness condition

An assembly consists of a set X and a nonempty set of natural-number realizers for each $x \in X$. A morphism is a function tracked by a partial recursive index that is defined on every valid input realizer and returns a valid output realizer. In the standard natural-number assembly, take the realizer of n to be n itself. Every endomorphism of this assembly therefore has a total recursive underlying function, directly by its tracking condition. Conversely, a total recursive function supplies a tracker. This is the precise effectiveness property needed here; realizability toposes extend this setting [12].

The displayed source operations have computable trackers. Structural compilation is computable, and the preservation theorem transfers the source tracker to the machine presentation. A bare topos model or a truth value for $\exists y$ supplies no such tracker. One should not claim a semantics-preserving compiler merely because a statement has an interpretation in an effective topos; the operational relation and tracking proof still have to be constructed.

8 Higher-topos and homotopy semantics

8.1 The boundary imposed by Lean's equality

Lean's equality is proposition-valued and its proofs are definitionally proof irrelevant; equality proofs are consequently unique [7]. Stock Lean is therefore not being identified with a type theory whose identity types retain arbitrary higher homotopies. A faithful identity-preserving interpretation of *all* homotopy types into such proof-irrelevant equality would collapse nontrivial path structure. The present higher semantics is external, or may be encoded through explicit mathematical objects in Lean. It is not obtained by adding univalence as a computationally inert axiom to the extracted program.

8.2 The arithmetic core in a higher topos

Let $\mathcal{X}$ be a Grothendieck $(\infty, 1)$-topos with its natural-numbers object $N_{\mathcal{X}}$. Interpret context n as $N_{\mathcal{X}}^n$. Numerals, projections, addition, numerical case distinction, parameterized primitive recursion, and composition interpret the source. The source and compiled machine equations give the same morphism. Since these context objects are 0-truncated, their mapping spaces are 0-truncated. Thus this arithmetic core does not acquire new nontrivial higher path data merely by changing the ambient category.

The point of the ambient higher topos is to express families, refinements, and changes of presentation coherently. For a general family $B : A \to \mathcal{X}$, one distinguishes

$$\prod_{x:A} B(x) \qquad \text{from} \qquad \prod_{x:A} \|B(x)\|_{-1}\,.$$

In the slice $\mathcal{X}_{/A}$, the latter asserts that the (-1)-truncation of $B \to A$ is terminal: the map is an effective epimorphism. The former is a section. There is no general operation recovering a section from the truncated family. This is the homotopical counterpart of the topos counterexample above [15].

Proposition 8.1 (Unique fibers, with an essential qualification)**.** *If $p : B \to A$ is both (-1)-truncated and an effective epimorphism in an $(\infty, 1)$-topos, then p is an equivalence and has a section, unique up to a contractible space of choices.*

Proof. Its fibers are propositions and are locally inhabited. The factorization into an effective epimorphism and a monomorphism forces a map in both classes to be an equivalence. Equivalently, each propositional fiber has contractible truncation and is already its own (-1)-truncation, hence is contractible. The space of inverses to an equivalence is contractible. □

This does not contradict the halting example. That example is classically single-valued, but no effective presentation of its totality and fibers has been supplied in a computational model. Semantic uniqueness inside a suitable constructive model and classical uniqueness of an externally given relation are different hypotheses.

8.3 Compilation coherence and localization

For a sequence of representations L_0, L_1, L_2, let C_{01}, C_{12} be translations and let η_{ij} be semantic-preservation witnesses. Their composition must agree with the witness for $C_{12}C_{01}$, up to the equality or homotopy notion in the chosen semantic category. Natural-number observations reduce this obligation to pointwise equality; richer representations require explicit higher coherence data.

More formally, choose a small category $\mathcal{P}$ of operational presentations and simulation maps, a category $\mathcal{D}$ of observable behavior, and a functor $O : \mathcal{P} \to \mathcal{D}$. Let W be the maps sent by O to isomorphisms. By the defining universal property of $(\infty, 1)$-categorical localization [14], the induced functor on nerves factors as

$$N\mathcal{P} \longrightarrow N\mathcal{P}[W^{-1}] \xrightarrow{\overline{O}} N\mathcal{D},$$

with the appropriate contractible uniqueness of the factorization. This is a semantic organization principle, not an algorithm for finding simulations. Each proposed compiler pass must first supply its simulation and observation-preservation proof. Runtime cost or memory can be retained as observations; if only output values are observed, cost equality is not obtained.

Existence of strict univalent universes in presentations of Grothendieck higher toposes supports internal homotopy type theory under specified hypotheses [16]. It does not establish executable canonicity of arbitrary Lean terms. Cubical canonicity results depend on additional computational structure, and even distinguish judgmental from homotopy canonicity [17]. Runtime proof erasure is neither automatically a truncation modality nor automatically the inverse-image part of a geometric morphism. Those identifications would require preservation and adjunction properties not asserted here.

9 Verification by different kernels

As of the pinned Lean 4.35.0 release, `lake check -paranoid` runs the official checker and the additional bundled checkers `leanchecker-paranoid`, `lean4lean`, `nanoda`, `con-leche`, and `con-ron` [8, 9]. The hardened official checker is a diversification of the same kernel, not a separately invented logic. `lean4lean` closely follows the official implementation. `con-ron` is related to `con-leche`. Kernel counts are consequently not counts of statistically independent experiments.

The published con-leche consistency statement is relative to a model of a specified set theory; it is not an assumption-free proof of Lean's consistency. Its metatheory can use classical assumptions even when the *object program being checked* has an empty axiom closure. Likewise a checker correctness theorem, its executable implementation, and the toolchain that built it are different trust layers [10]. No assurance claim here assumes that these layers have disappeared.

9.1 Release predicate

Let A be the candidate artifact, S the externally fixed specification, X the exported declaration stream, and $K_1, \ldots, K_r$ the pinned checkers. Define

$$\begin{aligned}\mathsf{Release}(A, S, X) \iff \mathsf{BoundTo}(A, S, X) \;\wedge\; \mathsf{Ax}_D(\mathsf{roots}) = \varnothing \\ \wedge\; \mathsf{Supported}(A) \;\wedge\; \mathsf{CertificatesMatch}(A, S) \\ \wedge\; \bigwedge_{j=1}^{r} \mathsf{Accept}_{K_j}(X) \;\wedge\; \mathsf{EvidenceComplete}(A).\end{aligned}$$

Here BoundTo includes exact source and specification identifiers, generated-code identity, module roots, and checker input identity. EvidenceComplete includes complete commands, versions, hashes, modes, exit codes, and logs. A timeout, a declined feature, a missing checker, or an internal error does not count as acceptance. The release uses conjunction, not majority vote. A disagreement blocks release and is investigated; no checker is dropped merely because it disagrees.

Proposition 9.1 (Conditional multi-checker assurance)**.** *Suppose the binding and audit procedures are correct, and at least one accepting checker is sound for the exported fragment and its stated*

assumptions. Then the exported correctness theorem satisfies that checker's soundness guarantee. If the guarantee is derivability, the theorem is derivable; if it is model existence or model truth, the conclusion is limited accordingly.

Proof. The release predicate implies acceptance by each checker, hence by the sound one. Apply its soundness statement to the exact common input. Binding identifies the theorem with the intended declarations. The audit removes declared axioms from the claimed roots; it does not remove hypotheses or foundational metatheoretic assumptions. □

Agreement alone does not prove that one of the checkers is sound. No numerical failure probability follows without justified assumptions on correlated defects. Different compilers, runtimes, and implementations reduce some common failure modes, but shared semantics, exports, or specifications remain common dependencies.

9.2 An exact axiom audit in addition to `lake check`

The built-in `lake check` permits three standard axioms: propositional extensionality (`propext`), choice (`Classical.choice`), and quotient soundness (`Quot.sound`). Its success therefore does *not* establish the strict profile. Run `#print axioms` for every manifest root during development, and require a structured closure audit with an empty allowed set at release. Audit the program, correctness and compiler theorems separately, so an accidentally axiom-dependent proof is visible even if executable code is clean. Merely importing a library that contains unused axioms does not mean a selected root uses them; report both whole-export contents and root-reachable dependencies without confusing them.

10 Lean implementation contract and milestones

The implementation should separate pure definitions and proofs from drivers that read files, invoke checkers, and write logs. The following signatures specify the intended shape; they are interface notation rather than a claimed compiled module:

```
Expr       : Nat -> Type
DataExpr   : Nat -> Type
Code       : Nat -> Nat -> Nat -> Type
Env        : Nat -> Type
Stack      : Nat -> Type
eval       : Expr n -> Env n -> Nat
erase      : Expr n -> DataExpr n
compile    : DataExpr n -> Code n h (h + 1)
run        : Code n h k -> Env n -> Stack h -> Stack k
erase_correct : forall e rho, evalData (erase e) rho = eval e rho
compile_correct : forall e rho s,
  run (compile e) rho s = push (evalData e rho) s
```

Raw bytecode is separately parsed and checked into `Code`; error cases belong to that parser/checker, not to the well-typed interpreter. These dependent indices are an implementation convenience and require no choice. Pointwise statements avoid unnecessary function extensionality. Each eventual declaration is accepted only after its actual axiom list has been checked; a preferred tactic name is not an axiom guarantee.

Stage	Deliverable	Completion condition
M0	Frozen syntax, machine, input/output contract, and version policy	Human review of examples and non-goals; specification hashes recorded.
M1	Lean source and target syntax, total evaluators, scoping/effect checker	Constructor coverage; checker soundness; no `sorry`; empty axiom closure of selected roots.
M2	Erasure and structured stack compiler	Mechanized erasure, stack-effect, frame, and compiler-preservation theorems.
M3	Frontend translation validation and declaration audit	Checked bridge to exact source; sound closure traversal; malformed and unsupported inputs rejected.
M4	Serialization and executable driver	Decode/encode round-trip; input bounds; code identity bound to theorem; resource outcomes explicit.
M5	Diverse-kernel release pipeline	All selected checkers accept the same bound export; empty root-axiom audit; independent rebuild evidence.
M6	Optional native backend and richer source fragment	Separate target semantics and refinement proofs; no extension inherits verification without its obligations.

The foundational correctness proofs in M1–M3 should use structural induction, primitive recursion, equality substitution, and explicit pointwise relations. Avoid quotienting syntax or proving function equality by `funext` when setoid laws suffice. The richer categorical and higher semantic metatheory may be developed in a separate project with an explicitly different axiom policy; its assumptions must not enter the executable compiler's audited theorem closure unnoticed.

10.1 Required evidence and adversarial cases

The release evidence contains the specification and source digests, exact Lean version, dependency lockfile, target root list, full theorem statements, axiom closures, export digest, each checker's binary digest and mode, exit status, and complete logs. It also records the bytecode and executable digests, test seeds, outcome counts, and the unverified boundary. A source commit alone does not identify a compiled binary.

Positive tests cover zero and positive loop bounds, lexical shadowing, both branches, nested loops, large natural values, and programs with nonempty underlying stacks. Negative tests cover out-of-range variables, stack underflow, mismatched branch effects, invalid scopes, undisclosed declarations, axiom dependencies hidden in theorem bodies or typeclass instances, unresolved hypotheses, incorrect theorem statements, and corrupted serialized code. Mutation tests deliberately change one translation clause, variable index, loop order, or annotation treatment and require rejection or a counterexample.

Finite tests complement proofs and check the surrounding implementation. They cannot establish the universal compiler theorem. A timeout must remain an observed incomplete run, not an incorrect theorem or successful validation. Performance tables are published only after measurement with exact hardware, versions, and resource limits; none are fabricated for a future implementation.

10.2 Executed reference-model validation

The accompanying standard-library Python model was run with Python 3.12.14 and seed 20261027. It enumerates all 560 well-scoped source trees of sizes one through four, with literals zero and one, one opaque annotation label, and contexts of lengths zero through two. Environments range over $\{0, 1, 3\}^n$. Across four initial stacks this gives 3,456 source/environment cases and 13,824 stack cases. An additional 1,000 generated programs of depth at most four give 4,000 stack cases. Eight targeted programs, 14 invalid-input rejection cases, two deliberately corrupted translations, one arbitrary-code base-stack carry case, and explicit budget exhaustion also pass. The base-stack case checks the distinction between height preservation and the compiler's stronger frame theorem.

These are finite behavioral checks of the executable specification. Annotation labels stand for proof-irrelevant payloads; Python does not check a Lean proof. The report records source hashes and explicitly records that no full Lean proof or independent-kernel validation was performed for this reference model. No measured runtime-speed comparison is claimed.

11 Threat model and limitations

The adversary may be a faulty or malicious proof-producing agent, a frontend that emits the wrong theorem, malformed exported terms, a compromised build step, or a buggy checker. The design uses a fixed specification, proof-producing translation, semantic theorem binding, independent replay, and reproducible evidence. It does not assume that an agent's confidence or a successful command is a correctness certificate.

The compiler theorem addresses functional behavior in an ideal natural-number machine. It does not establish constant-time execution, memory bounds, confidentiality, fairness, liveness of external services, correctness of an operating system, or resistance to hardware faults. The pure arithmetic subset cannot express every terminating Lean function. Expanding the fragment is a research and implementation task, not a matter of relaxing a blacklist.

Most importantly, *checked by different kernels* is not equivalent to *has no bugs*. A correct proof of a mistaken specification passes every sound kernel. A correct source-level proof can coexist with a defective parser, backend, build driver, runtime, or deployment. The defensible claim is exact: specified properties were verified for identified artifacts relative to a declared foundation and implementation boundary, and the same proof artifacts were accepted by the listed kernels.

12 Related work and publication status

Lean combines dependent type theory with executable programming facilities [1]. Its existing compiler erases proofs, but this manuscript's proposed compiler and validation procedure are separate artifacts. Certified erasure in MetaCoq demonstrates the need to prove preservation for a precise source theory and reduction semantics; that work is not automatically a correctness proof for Lean's compiler [11]. Realizability supplies an effective interpretation, while homotopy type theory and higher-topos models distinguish witnesses, truncations, and coherent semantics [12, 15, 16, 17].

This is an anonymous, human-directed manuscript prepared with AI assistance. Responsibility for submission and the final claims remains with the submitting author or research group; any required private editorial identity should be supplied through the journal's authorized process. The paper and reference model are prepared for review. The full Lean implementation and the diverse-kernel release evidence remain the next stages. This manuscript does not report completed journal submission, acceptance, or publication.

A Practical Lean interfaces

The following ordinary Lean pattern explains the witness/certificate boundary:

```
def successorWitness (n : Nat) : {m : Nat // m = n + 1} :=
  Subtype.mk (n + 1) rfl

def successorProgram (n : Nat) : Nat :=
  (successorWitness n).val

theorem successorProgram_correct (n : Nat) :
    successorProgram n = n + 1 := rfl
```

This small example uses ordinary Lean syntax, but is illustrative and is not presented as a completed compiler implementation or as checked evidence. Audit the exact declarations with `#print axioms`; a reader can then compile a driver separately while retaining the runtime trust boundary.

B Reproduction and submission bundle

The accompanying package contains the complete LaTeX source, PDF, compact journal-body text, structured implementation requirements, and a standard-library-only executable reference model. The model exercises the exact binder order and stack framing of Sections 4–5; it is not the future Lean implementation. Its test report states the executed finite test set. The publication metadata keeps the public author anonymous and makes no terms-acceptance or submission claim.

The journal's inspected submission form permits a 140-character title, a 400-character abstract field, and a 20,000-character Markdown/LaTeX body, with associated files. The compact body is designed for those limits; the PDF carries the full treatment. Submission and any agent enrollment are distinct from manuscript preparation.

References

[1] L. de Moura and S. Ullrich. The Lean 4 Theorem Prover and Programming Language. *CADE 28*, 2021, pp. 625–635. doi:10.1007/978-3-030-79876-5_37.

[2] Lean documentation. *The Lean Language Reference: Axioms*. https://lean-lang.org/doc/reference/latest/Axioms/. Accessed 27 October 2026; live documentation may advance beyond the pinned toolchain.

[3] Lean documentation. *Definitions: Modifiers*. https://lean-lang.org/doc/reference/latest/Definitions/Modifiers/. Accessed 27 October 2026.

[4] Lean documentation. *Recursive Definitions*. https://lean-lang.org/doc/reference/latest/Definitions/Recursive-Definitions/. Accessed 27 October 2026.

[5] Lean documentation. *Elaboration and Compilation*. https://lean-lang.org/doc/reference/latest/Elaboration-and-Compilation/. Accessed 27 October 2026.

[6] J. Avigad, L. de Moura, S. Kong, S. Ullrich, et al. *Theorem Proving in Lean 4*, Chapter 12: Axioms and Computation. https://lean-lang.org/theorem_proving_in_lean4/Axioms-and-Computation/.

[7] Lean documentation. *Propositional Equality*. https://lean-lang.org/doc/reference/latest/Basic-Propositions/Propositional-Equality/. Accessed 27 October 2026.

[8] Lean project. *Lean 4.35.0 release notes*, 12 October 2026. https://lean-lang.org/doc/reference/latest/releases/v4.35.0/. Pinned implementation source: https://github.com/leanprover/lean4/tree/v4.35.0.

[9] L. de Moura. *Lean Now Ships with Verified Kernels*, 15 October 2026. https://leodemoura.github.io/blog/2026-10-15-lean-now-ships-with-verified-kernels/.

[10] J. Breitner and contributors. *The con-leche proof idea*, revision 24980d94f2, 23 October 2026. https://leanprover.github.io/con-leche/con-leche-proof-idea.html. The cited page links the formal theorem and its hypotheses.

[11] M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, and T. Winterhalter. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proc. ACM Program. Lang.* 4 (POPL), Article 8, 2020. https://metarocq.github.io/coqcoqcorrect.

[12] S. Awodey, L. Birkedal, and D. S. Scott. Local realizability toposes and a modal logic for computability. *Mathematical Structures in Computer Science* 12(3), 2002, pp. 319–334. doi:10.1017/S0960129502003675.

[13] B. Jacobs. *Categorical Logic and Type Theory*. Studies in Logic and the Foundations of Mathematics 141, Elsevier, 1999. https://www.cs.ru.nl/B.Jacobs/CLT/bookinfo.html.

[14] J. Lurie. *Higher Topos Theory*. Annals of Mathematics Studies 170, Princeton University Press, 2009. Author's preprint: https://arxiv.org/abs/math/0608040.

[15] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*, 2013. Sections 3.7–3.8 on propositional truncation and choice. https://homotopytypetheory.org/book/.

[16] M. Shulman. *All* $(\infty, 1)$*-toposes have strict univalent universes*, 2019. arXiv:1904.07004.

[17] T. Coquand, S. Huber, and C. Sattler. Canonicity and homotopy canonicity for cubical type theory. *Logical Methods in Computer Science* 18(1), Article 28, 2022. doi:10.46298/lmcs-18(1:28)2022. https://lmcs.episciences.org/9043.

[18] Meta Journal of Agentic Meta Algorithms. Submission guidance and agent protocol. https://agentic-meta-journal.pages.dev/submit; https://agentic-meta-journal.pages.dev/agents. Inspected 27 October 2026.