Solidity Semantics for Compiler Verification: Effect Orders, Physical Aliases, and Observations

OKcontract Labs
Lean formalization for Solidity 0.8.36 and Cancun
12 November 2026

***Abstract*—Compiler verification requires source semantics that retain observable effect-order choices, aliasing, and rollback while keeping proof and resource boundaries explicit. We present Solidity Spec, an executable Lean development targeting Solidity 0.8.36, its IR compiler pipeline, and Cancun. Its resolved continuation machine selects eager child expressions independently, operates over a shared physical aggregate heap, and distinguishes source failure from unsupported behavior and host exhaustion. The architecture separates checked entry points, fixed implementation policies, and compatibility imports. The release combines explicit theorem contracts, admission counterexamples, independent compiler and EVM checks, and quantified source/target pilots, including an order-sensitive success-versus-rollback result. External execution checks assess agreement with Solidity implementations, while configuration invariants establish admitted-state safety for a restricted profile. This paper documents the implemented semantic domains and their proof obligations; complete Solidity admission and compiler preservation remain open.**

I. Research question and contributions

We ask: *How can executable Solidity semantics support mechanized compiler-preservation proofs while retaining permitted effect orders, physical aliasing, and an explicit distinction between source and model observations?* The central difficulty is compositional. A call conversion may copy an aggregate after argument side effects; a captured alias may observe a sibling write; a subsequent revert must erase that write. Proving safe execution requires an invariant that connects these operator rules to the heap, call frames, and failure behavior.

For example, let two initialized memory references `a` and `b` alias the same word array, initially containing y at index zero. In `(a[0] = x) + b[0]`, evaluating the read before the assignment gives $x + y$, while evaluating it afterward gives $2x$, provided the respective addition fits in a word. Checked overflow instead produces a source panic. A model used to justify transformations must preserve the permitted choices, the shared physical cell and the failure behavior together.

The release provides three concrete interfaces. First, a single continuation machine combines per-group child selection with captured physical places and an invocation checkpoint. Second, a resolved admission boundary retains the admission check equations; independent type, heap, and continuation invariants support progress, preservation and resource-adequate prefix safety for a stated profile. Third, quantified source/target pilots demonstrate calls with aliases and rollback, and two permitted orders that can disagree on success versus overflow. The second pilot derives machine execution and two concrete target traces for every input word, failure flag and related entry state. An enforced theorem manifest binds statements, axiom allowances and definition sources. Independent compiler and EVM checks provide empirical evidence about fourteen selected interactions. Compatibility bridges connect the certificate interfaces to the shared execution model.

Our work builds on executable Solidity semantics, memory models, and proof-assistant verification [2], [3], [4], [6], [7], [10]. We combine independent child scheduling, physical aliasing, and rollback in interfaces for compiler-preservation proofs. The quantified pilots test how these interfaces compose; comparison with prior developments and external execution checks place those results in context.

II. Which specification is being used?

The operational meaning of a proof in this development is fixed by the Lean definitions in its dependency closure. The relevant specification is therefore the executable code described here, together with the theorem's explicit premises. Solidity documentation and Ethereum fork rules guide the source and target definitions, whose agreement with those references remains a model-adequacy obligation, assessed through review and differential execution.

There are three principal layers. First, `Contract/Language` and `Contract/Semantics` define an intrinsically typed contract language. Second, `Solidity/` defines the source models used by the Solidity artifact certificates. Third, `Contract/Evm` gives partial bytecode interpreters and call/creation frames. Yul word expressions and a richer interaction model support these layers, with correspondence theorems connecting their supported domains.

Two source boundaries must be distinguished throughout. The *resolved machine* consumes typed syntax, declaration identities and a resolved function table; it supports a broader language, including loops and internal recursion. The *artifact certificate interfaces* relate models of a restricted source language to particular target artifacts. Clients establish source admission and exact artifact binding through external parsing and artifact capture. These trusted steps connect the source files and compiler output to the models in the theorems. The safety profile further restricts the resolved machine to the configurations covered by its invariants. Section XIII collects the trust assumptions for using these interfaces.

Solidity Spec is a standalone project at `okcontract/solidity-spec` [1]. It follows our work on the oksolc compiler. All paths in this paper are relative to this package unless explicitly qualified. Primary declarations use the `Solidity` namespace; certificate interfaces also use `Compiler.Validation` through compatibility imports. The release is pinned to Solidity 0.8.36, reference compiler identity `0.8.36+commit.8a079791`, Cancun, and Lean 4.29.0. Compiler compatibility is scoped to `viaIR: true`, with optimization disabled or enabled at 200 runs; legacy code generation is outside this profile. `target.json` and `Target.lean` record those targets. Versioned documentation provides explanatory references [12], [14], [15], [16]. Although the documentation site may label its 0.8.36 pages "develop," the exact native compiler identity pins differential tests. Accepted pragmas retain this fixed semantic target.

A. Compiler-pipeline boundary

The choice of compiler pipeline affects the timing of implicit copies. With an initially empty storage array `values`, consider:

```
values.push(1);
Holder memory h =
    Holder(values, append());
return h.a.length;
```

Here `Holder` has a memory-array member, and `append()` pushes 2 onto `values` before returning zero for its other member. Legacy code generation copies the first argument before evaluating the second, producing length one. IR evaluates the raw arguments before copying, producing length two. The corresponding internal call with a memory-array parameter behaves the same way. Both results were reproduced by the external driver with the pinned compiler, Cancun, and both optimizer settings. The indexed fixture returning `h.a.length*256+h.a[1]` panics with code `0x32` under legacy and returns 514 under IR.

The construction and invocation rules select the IR conversion phase. Reproducing the legacy copy timing would require changing conversion placement as well as selecting an argument order. Explicit conversion nodes could support such an elaboration in future work. Solidity documents further differences between the pipelines [13], motivating the explicit compatibility profile used here.

III. The continuation machine and its safety contract

Let τ range over resolved scalar, aggregate-reference and function types, and let P be a finite table of typed function bodies. Local cells contain typed values or explicit uninitialized references; heap references identify physical locations rather than copied mathematical arrays. The execution layer defines closed resolved syntax, heterogeneous value packets and one continuation machine. Stable local IDs and captured physical places retain the location selected by effectful access paths. A memory-member assignment rebinds its captured pointer cell; readback reloads that cell. Nested tuples and holes remain groupings, without fabricating a storable tuple value.

A configuration is $c = (t, K, s, s_0)$: a current task t, continuation stack K, current machine state s, and invocation checkpoint s_0. The state contains typed local cells, arithmetic mode, the physical heap and world, external backend state, and remaining materialization work. A task may evaluate an expression, capture a place, collect operands, invoke a function or resume control. One transition selects a permitted child when needed and updates these components; an invocation failure restores s_0.

Each eager expression group selects children independently. Independent list-split selection rules prove that every occurrence in a completed group is evaluated once and that every permitted permutation has a schedule; indexed results return to syntactic order afterward. Boolean operators and conditionals are lazy. Assignment captures its destination once. Conditional arms carry their resolved common type; an implicit tuple-group transfer runs after the whole selected arm has evaluated, so later argument effects precede aggregate copies. Compound assignment permits the old read before or after the RHS. Tuple stores use reverse component order, whereas explicit return conversions follow declaration order. These parent operations are distinct from unspecified child-expression order.

Inline arrays allocate backing memory before evaluating their elements. Struct constructors evaluate all raw arguments before root allocation, then convert and store fields in declaration order. Constructor packets permit shared reference children, allowing aliases within a constructor packet. Exiting a lexical scope removes introduced local cells while retaining writes to outer variables. All loop forms, break/continue and return share the same transition vocabulary.

Internal calls use a finite signature/body table and support recursion, reference parameters, named returns and multiple results. Raw argument effects precede parameter conversion, which precedes indirect dispatch and its null-handle panic. Callees start checked, share allocation and the invocation heap, and restore caller locals and arithmetic mode. Primitive store codecs retain the certificate logical-store domains, including dirty-Boolean cleanup, without assuming a universal physical-slot injection.

Source failures roll back the complete invocation checkpoint. Unsupported or invalid contexts and undefined behavior are explicit inconclusive observations; host exhaustion retains its configuration. Continuation bounds limit host evaluation; EVM operand-stack and gas bounds are properties of a bound target artifact. Finite soundness/completeness covers terminating runs, and an infinite loop witness demonstrates divergence within the same semantics.

Whole-tree resolved admission checks scopes and source permissions, including unused branches. Declaration and mutability resolution, and callee summaries of control flow where required, are inputs to these checks. General source-byte admission, aggregate event/error ABI encoding, arbitrary raw-memory correspondence and compiler lowering preservation remain separate obligations. The interfaces are defined under

TABLE I
COVERAGE BY KIND OF EVIDENCE WITHIN THE STATED RESTRICTED REPRESENTATION/ADMISSION PROFILES. EXTERNAL CHECKS COVER SELECTED MANUALLY PAIRED IR CASES.

Construct	Represented	Admitted	Generic safety	Target refinement	External check
Scalar operations and word storage	Yes	Yes	Profile	Selected pilots	Selected cases
One-level `uint256[]` memory	Yes	Yes	Profile	Prepared entries	Selected cases
Struct construction and aggregate copying	Yes	Yes	No	No	Selected cases
Internal calls and reference parameters	Yes	Yes	Profile	Alias/call pilot	Selected cases
Events and their rollback	Yes	Yes	No	No	One rollback case
General source-to-resolved elaboration	No	No	No	No	No

`Solidity/Core/Execution`.

A. From admission to represented configurations

Soundness and completeness of fuel-bounded iteration relate the runner to the step semantics, whose terminal observations include inconclusive results. Admitted-state safety additionally requires an invariant connecting syntax, local values, captured targets, saved continuations, function handles, and physical heap packets. The safety proof shows that transitions preserve this invariant under explicit domain and primitive-resource assumptions.

The proved structural profile includes executable scalar types, closed recursive function handles, canonical memory word arrays with physical aliases, scalar persistent and transient places, eager effects, structured control and singleton declaration prefixes. The independent `ConfigurationRep` predicate connects checking equations to initialized cells, heap packets, operand partitions, captured targets and saved continuations. These structural premises allow the proof to derive progress. With adequate primitive resources, `Machine.step` either produces a represented source observation or preserves the invariant and checkpoint, excluding inconclusive results and primitive suspension. Entry requires successful admission, the structural profile, a closed admitted function table, and an initial state with represented physical packets and closed function handles.

The profile excludes other aggregate families, storage/calldata reference locals, tuple-pattern assignments, inline-array and struct construction, array-length mutation, event/error encoding, assembly and linked calls. User-defined value wrapping/unwrapping, external-function projections and `delete` are also excluded; a for-loop post is an expression or skip. `SAFETY.md` indexes the declarations defining this profile within the broader executable machine.

Resource adequacy is separate from termination. Aggregate fuel is at least two and host continuations are uncapped. Let $C = 66(2^{256} - 1) + 96$. The generic progress theorem assumes at least C remaining host work; an initial allowance of $(n+1)C$ suffices for an n-transition prefix. These conservative bounds suffice for the generic proof; particular executions can use less work. The extra allowance covers the runner's real terminal inspection even at zero transition fuel. The `execute_prefix` theorem covers every child-choice sequence. A completed prefix has represented values or exact source-failure rollback; an exhausted prefix retains the invariant, original checkpoint and a further C allowance. The bound measures modeled materialization work; EVM gas and physical host capacity require separate analyses. Final packet typing is existential in the terminal predicate, so the generic theorem establishes representation preservation and absence of model failure. Relating those packet types to the root expression's result type remains open. The pilot theorems establish the exact returned words for their particular programs.

Two regression families illustrate the distinction. Invocation performs the permitted implicit internal-function conversion, including pure-to-view widening, while retaining the handle. Separately, eager siblings receive the same incoming reference-readiness facts. Assignments guaranteed by a complete group become available afterward; a read requiring another sibling to run first is unsupported by this defined admission domain. Lazy operands and sequential statements retain their required order. The raw machine still represents the original order-dependent inconclusive schedules.

The lightweight import `Solidity.Execution` exposes checked entries, the machine and its execution facade. Its `Proofs` submodule provides additional execution laws, and `Safety` exports admitted-state safety. Compatibility clients use `Solidity.Compatibility.Execution`; the comprehensive `Solidity` umbrella includes all four interfaces. Indexed checked entries retain the admission equations. Their projection proofs preserve the facade, including failed preparation. `Policy.lean` defines fixed compiler choices, with source provenance recorded in `references/policies.json`. These interfaces share the same normalizer and interpreter.

IV. OBSERVATION CONTRACT AND PROVENANCE

We distinguish three observation profiles. A *functional* observation compares completed return/revert data and the selected public state and effects. A *metered* observation additionally compares the specified resource outcome. A *representation-sensitive* observation also binds the exact code or packet representation required by its relation. These profiles are recorded in `SEMANTIC_CONTRACT.md`, with explicit premises for each relation. `Observation.lean` defines completed records and named relations over invocation outcomes, public state, transient storage, account lifecycle and ordered interactions. Stronger profiles also compare resource observations and the bound code representation. Correspondence requires com-

pleted, supported execution and proof of any capability restrictions stated in the summary.

Invocation rollback restores the entry snapshot, including logs and transaction-local storage. Gas consumption and transaction admission/fee settlement have separate boundaries. The current profile starts after admission and intrinsic gas accounting, with fee settlement handled externally. Account existence and Cancun creation/deletion facts are tracked explicitly alongside balances and code.

Sixty-two reference snapshots are pinned by immutable upstream revision and content digest in `references/lock.json`. They include selected Solidity documentation and type rules and the Cancun execution-specs fork directory. The lock distinguishes complete snapshots from documentation indexes with unbundled chapters, making the reference corpus inspectable at a fixed revision.

V. Quantified source/target experiments

A. Internal calls, aliases and rollback

The composed pilot quantifies over input words x, failure flags b and related initial states. Two initialized memory references alias one represented length-one word array. An ordinary entry call invokes an internal helper, which writes the array payload and persistent slot zero. The caller reads through the other alias and either returns x or reverts. The source theorem derives parameter binding, named-result initialization, both calls, result transfer and caller restoration through the continuation machine. It establishes a checked completion within 160 transitions with choices `[]` for every materialization allowance.

The target is a manually paired, hand-authored 23-byte Cancun fragment. Its prepared-entry relation gives six active memory words and zero bytes beyond that extent. A metered target trace determines adequate-gas observations; failure and insufficient gas restore the complete target world and transaction snapshot. Frame theorems preserve other owners, other storage slots and memory packets disjoint from the written payload. The proof derives both the source run and its correspondence with this trace from the entry-representation premises.

`refines_represented` composes this same budgeted source execution with generic safety at $B \geq 161C$, then derives matching adequate-gas target observations and separate insufficient-gas rollback. Both exported refinements retain $g_{\mathrm{remaining}} = g - used$ for every $g \geq needed$, with $used \leq needed$. Their source/target relation is functional; the equation additionally specifies target gas consumption, without assigning EVM gas to the source. A nonemptiness theorem constructs its represented, related domain for arbitrary persistent worlds, owners, words and flags. A negative fixture shows that the default 4096 work allowance suffices for this nonallocating trace but falls below the generic uniform bound. A companion applies all-choice prefix safety to the effectful aliased group `(a[0] = x) + b[0]`, permitting order-dependent values and source overflow panic.

Both experiments start from prepared entry states and hand-authored artifacts. Their scope is execution from those entries; source-byte correspondence, compiler emission, and deployment/allocation prefixes remain external to the composition. `PILOT.md` indexes the proof modules and reproduction commands.

B. Order-sensitive success and rollback

The second experiment uses two initialized references to the same initially zero memory word and the following prepared function body:

```
stored = x;
uint256 z = (a[0] = x) + b[0];
if (fail) revert();
return z;
```

The source has two concrete permitted schedules. Reading the alias first gives x; writing first gives $2x$ if it fits, and panic `0x11` otherwise. For $x = 2^{255}$ and a false failure flag, one schedule returns 2^{255} and the other restores the complete entry state. The initial storage write therefore participates in the rollback argument.

For each schedule, the `source_execution` theorem derives machine execution with a transition allowance of 80 for every input word, failure flag and host-work allowance, including zero. The `source_permitted` theorem embeds that trace in the permitted-execution relation. Admission is checked for the prepared body and its expected word return binder. Independent premises require represented aliases, an initially zero payload and writable storage. A domain witness establishes these premises and related target states for arbitrary persistent worlds.

Two fixed, hand-authored Cancun fragments implement these schedules: 25 bytes for the read-first case and 111 bytes for the write-first case. Both receive the original word and flag and perform the storage write. The latter checks the high-half bound before doubling. Their guarded traces determine a gas threshold and consumption, including the storage sentry. The exported `refines` theorem proves functional correspondence for every adequate gas budget, the exact remaining-gas equation, and complete target world/transaction rollback for overflow, explicit revert and insufficient gas.

Negative controls reject a wrapping zero return at 2^{255} for both contracted schedules and reject a target that matches the empty revert bytes while retaining its write. These results compose order, aliasing and failure in one source/target argument. The body operates on existing aliases; internal calls are covered by the first pilot and aggregate deep copies by the external tests. Its allocation-free trace also gives a zero-work bound for this program, illustrating how a program-specific argument can sharpen the generic profile's uniform bound.

VI. Released proofs and executable checks

The release separates proof checking, model examples and external conformance. The package build elaborates both libraries and their regression modules. Build checks use four

Lean workers. Fresh kernel replay checks the comprehensive library, the call/resource, child-order and order-sensitive pilot roots and the scalar machine regression closure, including their imports. These checks concern the released Lean definitions and proofs. External compiler execution is reported separately below.

A. Theorem contracts and composition

`theorems.json` records 85 selected roots with fully qualified names, expected elaborated statements, complete kernel-statement hashes and permitted axioms. A statement hash binds references to named predicates. The manifest additionally records SHA-256 identities for the imported project definition closure, build inputs and auditor. The release command compares compiled statements, complete kernel-statement hashes, per-root transitive axiom allowances and this source closure before fresh replay. Changed definitions therefore fail the gate even if a theorem's displayed statement is unchanged. Recording new expectations is an explicit review operation and is never an acceptance run. Fresh replay then checks proof terms independently of elaboration.

The two pilots compose checked admission and independent representation premises with source execution, target execution, gas conditions, and rollback for all quantified inputs. Explicit bounds on source transitions, aggregate recursion, and materialization work ensure that these executions complete.

B. Model fixtures

The package includes manually paired and generated model fixtures under `Solidity/Conformance`. The scalar corpus contains 45 scalar, loop and constant cases and ten constant rejection examples. The inference corpus contains 38 expression cases and 16 rejection examples; the static matrix contains 17,690 generated conversion, operator and update cases. These modules provide executable model expectations and source representations.

The aggregate corpus contains 46 runtime examples, five static rejection examples and nine model-boundary checks. The effectful corpus contains 48 machine programs, three source-rejection examples and two host-boundary checks. The interaction corpus contains 42 generated combinations of aliasing, evaluation order, conversions, calls and rollback. Expected outcomes are computed by executing the model, with manually paired source representations. These full-corpus counts describe model fixtures; the following external evaluation independently executes a selected subset.

Kernel regression modules exercise ABI canonicality, payability, storage layout, stateful execution, constructors, admission rejection, rollback and host exhaustion. Import and declaration-dependency guards check that primary execution interfaces use the shared machine. These concrete regressions complement the universal theorem contracts and help localize changes in behavior.

C. Independent compiler/EVM evaluation

The standalone driver executes fourteen selected effect cases through the Lean machine and checks resolved admission, then compiles the paired Solidity with the exact reference compiler and runs the artifacts on a local Cancun Anvil instance. Both IR optimizer settings are checked. The 28 observations matched exported permitted outcomes: 20 returned and eight reverted. There were no unexpected compiler rejections, model admission failures, unsupported executions, host exhaustions or observation discrepancies in this runtime set.

The comparison tests membership in a finite set of executed, permitted schedule witnesses. Because this set may cover only some schedules, a nonmember requires examining the pairing and coverage before diagnosing a semantic mismatch. The cases include aliased operand ordering, argument effects and copies, compound destination capture, overlapping tuple stores, internal reference calls, checked overflow, and rollback. Each call observes unchanged initial contract storage. Reverting cases additionally execute as transactions with failed receipts, empty logs and unchanged selected storage slots. The rollback check observes precisely these components of the transaction result.

Three separate static-rejection controls matched compiler diagnostic identities. Two model resource controls preserve exhaustion as a host limitation. The three copy-timing probes also run under legacy compilation with both optimizer settings: all six observations exhibited the documented boundary discrepancy and are recorded separately as negative compatibility controls. The runner retains exact source and Standard JSON inputs, compiler outputs, tool identities/hashes, RPC observations, process statuses and package input identities, allowing the corpus observations and compiler configuration to be reproduced. The manual source/model pairing remains a review obligation.

VII. Domains and semantic layers

Let $M = 2^{256}$, $\mathbb{W} = \{0, \ldots, M-1\}$, $\mathbb{A} = \{0, \ldots, 2^{160}-1\}$, and $\mathbb{B} = \{0, \ldots, 255\}$. Lean represents words and addresses using `BitVec 256` and `BitVec 160`. Bytes are `UInt8`; byte strings are finite lists. We write ρ for a local environment, σ for storage, and κ for context.

A. Word operations

`Word.lean` defines wrapping word operations and explicit checked operations. Wrapping addition, subtraction and multiplication operate modulo M. Logical shifts give zero for a shift count at least 256. The shared low-level unsigned division and remainder follow the EVM convention of yielding zero at a zero divisor; the checked variants below produce panic bytes.

Define $P(c)$ as the four selector bytes `4e 48 7b 71`, followed by 31 zero bytes and the one-byte code c. The checked arithmetic evaluator returns `Except (List UInt8) (BitVec`

256). For natural representatives a, b,

$$\mathrm{add}_c(a,b) = \begin{cases} \mathrm{ok}(a+b) & a+b < M, \\ P(11) & \text{otherwise}, \end{cases}$$

$$\mathrm{sub}_c(a,b) = \begin{cases} \mathrm{ok}(a-b) & b \le a, \\ P(11) & \text{otherwise}, \end{cases}$$

$$\mathrm{mul}_c(a,b) = \begin{cases} \mathrm{ok}(ab) & ab < M, \\ P(11) & \text{otherwise}. \end{cases}$$

Here panic codes are hexadecimal and the second branch denotes an error. Checked division and remainder use $P(12)$ at zero. Assertion failure in the typed language uses $P(01)$. These choices follow the intended checked-arithmetic and panic behavior [12]. Only addition, subtraction and multiplication from this checked family occur in the admitted Solidity word-expression grammar.

B. The typed contract language

Basic.lean defines types $\tau ::= \mathrm{word} \mid \mathrm{bool} \mid \mathrm{address}$. A variable is indexed by a type list Γ and its type; the matching Env stores values of exactly those types. Storage with n declared slots is a function $\mathrm{Fin}(n) \to \mathbb{W}$. Consequently, unbound variables, out-of-layout accesses, and ill-typed operands are unrepresentable in this AST.

Expressions include literals, variables, storage reads, caller/self addresses, word operators, comparisons and negation. Statements include returns, explicit reverts, local binding and assignment, checked operations, assertions, stores, branches, finite helper calls and event emission. Helpers are embedded closed bodies, giving a finite, nonrecursive call structure. Event topic count is bounded by four with a proof field.

The evaluator in Eval.lean is a total recursive function with the big-step interpretation

$$[\![s]\!](\sigma, \kappa, \rho) \in \mathrm{Success}(\sigma', \vec{v}, L) + \mathrm{Revert}(b).$$

Stores update one slot before evaluating the continuation. Helper results are prepended to the caller's environment; helper storage changes feed the continuation. Logs are ordered by execution. Revert carries no post-state and no retained logs. At the invocation boundary,

$$\mathrm{commit}_\sigma(o) = \begin{cases} \sigma' & o = \mathrm{Success}(\sigma', \vec{v}, L), \\ \sigma & o = \mathrm{Revert}(b). \end{cases}$$

Thus a store followed by revert has the same committed storage as immediate revert. The theorems revert_rolls_back and store_then_revert express this directly. Gas, ABI entry, and deployment are outside this typed evaluator. In particular, plain expression addition is wrapping; checked behavior is an explicit statement constructor.

VIII. The shared typed core

A. Values, operators and permitted orders

Scalar.lean indexes integer and fixed-byte values by $w \in \{8, 16, \ldots, 256\}$. Unsigned and signed integers share a bit-vector representation but have distinct types and operations. Signed interpretation uses two's complement. Checked addition, subtraction and multiplication test the corresponding mathematical range; unchecked operations wrap modulo 2^w. Signed division truncates toward zero. The minimum signed value divided by -1 panics in checked mode and wraps in unchecked mode. Division and remainder by zero always emit $\mathrm{Panic}(0x12)$, including within unchecked expressions. Arithmetic overflow emits 0x11. Shifts truncate without overflow checks; signed right shift sign-extends. The represented casts truncate or extend according to their typed constructor; these primitive operations are separate from source conversion permissions and literal elaboration.

Arithmetic mode belongs to each operator, so resolved syntax can represent nested checked/unchecked regions. Expression.lean, relative to Solidity/, defines literals, typed references, unary/binary operators, conditionals, conjunction and disjunction. References come from a resolved program. The executable evaluator chooses a uniform operand order. The separate relation Expr.Evaluates permits each binary node to choose independently; eval_permitted proves executable evaluation sound for that relation. The relation includes outcomes unavailable under either uniform strategy alone. Lazy constructs evaluate only the necessary operand/branch. These rules describe the retained pure fragment. The effectful machine in Section III connects its operand strategies to a shared heap and continuation stack.

B. Correspondence and stateful consumers

The modules Word.lean, TypedExpressions.lean, Conditions.lean, and Storage.lean (the latter three in the same directory) prove embeddings for arbitrary expressions, environments and current stateful source statement bodies. The typed DSL's zero-totalized EVM division is translated to a conditional that preserves its zero-divisor result within the shared core.

Statement.lean defines finite blocks over a typed store interface. The concrete compatibility store reuses source-derived field layout, ABI locals and static-write checks. Sequential composition propagates a terminal outcome; reverted and exceptional outcomes expose no committed store. body_result equates this evaluator with the certificate stateful evaluator. certified_contract then proves that a complete call certificate implies TracePreservesCore: the shared-core result conforms to the exact target trace for every sized input, store and static-mode setting, retaining the certificate's source domain and trace-derived gas observations. Typed helper/log statements and specialized ABI/constructor clients have explicit shared-core correspondence, extended below to the effectful machine.

C. Typed frames and certificate clients

The procedure layer reuses core expressions, environments and stores for finite closed helpers, local assignment, tuple returns and static-word logs. The theorem Compatibility.Typed.procedure_eval covers every statement of the retained typed language. Helper results extend the caller's frame; a later failure discards both helper and caller

logs and writes. The resolved effectful machine adds recursion; linked external execution remains a separate obligation.

Ordered declaration initializers and complete constructors execute core blocks, with proved outcome correspondence. Constructor bare returns retain their initialization meaning. Specialized ABI admission is related to the shared canonical-word predicate; empty, raw-word, literal and expression methods have core outcome adapters. Within the existing source domain, complete certificate transfers retain the calldata, static-mode, rollback, deployment and gas claims.

D. Structured control and lexical resolution

`Control.lean` adds a continuation machine for sequencing, branches, loops, break and continue. A loop carries a finite post block: skip for a while loop, or the increment for a for loop. Continue runs that post block, while break skips it. A return or revert bypasses all enclosing continuations. Finite interpreter execution is connected to a fuel-free transition judgment:

$$\mathrm{run}(k, c) = \mathrm{finished}(o) \quad \Longrightarrow \quad c \Downarrow o,$$

and every finite derivation has a sufficient interpreter budget. Separate theorems prove preservation under additional fuel and determinism for a fixed operand strategy. Exhaustion records the reached configuration as a host outcome, allowing finite-prefix reasoning for programs whose loops may diverge. Connecting these resolved loops to source text and target gas requires elaboration and artifact proofs.

`Resolve.lean` checks lexical references in typed scalar expressions and constructs de Bruijn variables. The nearest declaration wins; a type mismatch at that declaration is rejected rather than searching past it. Unknown names are rejected even in a branch skipped at runtime. Soundness and completeness relate the checker to an independent structural elaboration judgment. Evaluation preservation assumes agreement on resolved bindings. Parsing, type inference, overloads, inheritance and exact source-byte correspondence remain future frontend obligations.

E. Exact constants and checked preparation

`Literal.lean` evaluates constant-expression trees over normalized rational numbers. Addition, subtraction, multiplication, division, remainder, exponentiation, bit operations and shifts follow the pinned 0.8.36 rules. Remainder uses a quotient truncated toward zero; right shift of a negative integer rounds downward. Division by zero, fractional bit operands, invalid shifts, precision exhaustion and non-representable conversions produce static elaboration errors. A universal bit-observation theorem proves the Boolean meaning of signed constant bit operations, including every sign-extension bit.

The result precision guard checks the highest set-bit index of the normalized numerator and denominator against 4096. Exponentiation and left shift also apply conservative work guards. Thus 2^{2049} is rejected as a literal power even though its result would satisfy the separate result guard. The pinned compiler treats 0^e as zero for every nonzero literal exponent, including negative exponents. Such edge cases are explicit versioned rules.

Integer conversion is characterized in both directions. For unsigned width w, with q a normalized rational and v a w-bit word,

$$\begin{aligned}&\mathrm{toUnsigned}_w(q) = \mathrm{ok}(v)\\ &\quad\Longleftrightarrow\ q.\mathrm{den} = 1 \ \wedge\ \mathrm{toNat}(v) = q.\mathrm{num}.\end{aligned}$$

The signed theorem replaces the final observation by $\mathrm{toInt}(v)$. Both characterizations require exact representability of the constant, whereas runtime narrowing casts truncate to the target width.

`Traversal.lean` implements checked substitution of expression leaves and its structural soundness/completeness theorems. `Prepare.lean` uses it to resolve names and convert exact constant subexpressions. Its separate elaboration judgment requires lexical binding or integral, exactly representable constant values. The checker is sound and complete for that judgment and preserves evaluation when source leaves have their specified meanings. It checks all branches, including those skipped at runtime, operating on parsed trees with explicit operator and type annotations.

Normalization precedes the result-precision check. For example, adding two copies of $1/2^{2049}$ has a raw denominator 2^{4098} but a reduced denominator 2^{2048}. The reduced rational satisfies the result-precision guard, as demonstrated by kernel-checked examples in the released literal fixtures.

IX. The admitted Solidity source semantics

A. Resolved pure word expressions

The syntax in `Word.lean` is

$$e ::= n \mid x \mid {\sim}e \mid e \circ e, \qquad \circ \in \{+, -, *, \&, |, \oplus, \ll, \gg\}.$$

`Expr.eval` maps a name environment $\rho : \mathrm{String} \to \mathbb{W}$ to a word, with literal conversion modulo M. `Checked.lean` lifts evaluation into an exception monad, checking arithmetic and propagating exact panic bytes. Bitwise and shift operators retain their wrapping/truncating meaning [16]. The source model uses one checked flag for the entire body.

The AST is deliberately more permissive than the admitted source text. A natural literal in Lean can exceed the word range, and the environment is total. The trusted parser must establish literal bounds, name resolution, types and allowed constructs. It rejects literal arithmetic with Solidity's unbounded rational-literal semantics; operands instead require admitted word values such as `uint256(3)`. This prevents silently interpreting a rational constant expression as modular arithmetic.

Checked expression evaluation visits operands left to right. In this pure fragment, all arithmetic failures use the same panic bytes, making the eager evaluation order unobservable. Boolean short-circuiting in the stateful model is explicit and does affect whether an operand can panic [12].

B. Function inputs and observations

`Function.lean` records a name, word parameter names, body, checked flag and payability flag. Input consists of calldata d and incoming value v. Parameter i is decoded from byte offset $4 + 32i$ as a big-endian word. Successful word results are encoded as 32 bytes; arithmetic failure returns the panic bytes as a reverted outcome.

A signature's selector is the first four bytes of Keccak-256 over its canonical name and parameter types [14]; the implementation is `Keccak.lean`. The stronger call interface in `Call.lean` accepts a word method exactly when

$$\begin{aligned} A_f(d, v) \iff & (f.\mathit{payable} \vee v = 0) \\ & \wedge |d| \geq 4 + 32\, \mathit{arity}(f) \\ & \wedge \operatorname{selector}(d) = \operatorname{selector}(f). \end{aligned}$$

Failure of admission gives empty revert bytes. Trailing calldata is allowed. The separate `ValidCall` predicate requires exact canonical length and zero value. It witnesses inhabitation within the broader input domain of the all-calls certificate.

C. Typed ABI and dispatch

`ABI.lean` adds word, Boolean and `bytes4` parameters. A Boolean word must be zero or one; fixed bytes occupy the high-order bytes, with zero padding in the remaining 28 bytes [14]. In notation,

$$\begin{aligned} A_{\mathrm{bool}}(w) &\iff w \in \{0, 1\}, \\ A_{\mathrm{bytes4}}(w) &\iff w \,\&\, (2^{256} - 2^{224}) = w. \end{aligned}$$

Malformed complete argument words therefore revert during decoding.

The contract-level interfaces compose complete runtime-entry traces for short calldata, unknown selectors, disallowed value, truncated arguments, malformed arguments and selected body results. Selector uniqueness is a certificate condition. Separate models cover empty bodies, exact raw-word returns, literal Boolean/address/`bytes32` returns and empty contracts, with expression support limited to each model's grammar. The stateful source model also supports an optional receive body for empty calldata; general fallback handlers and external calls remain beyond this fragment.

D. Stateful statements and control

`StorageSource.lean` defines comparisons, Boolean conditions, statements and functions over a loop-free grammar. Statements assign a word or Boolean field, return a word, stop with empty return data, revert, or branch into finite statement lists. Conditions have literals, Boolean field reads, unsigned comparisons, negation, conjunction and disjunction.

Evaluation has an explicit continuation result:

$$\mathrm{Flow} = \mathrm{Next}(\sigma) + \mathrm{Halt}(\mathrm{Outcome}),$$

where an outcome is returned bytes plus storage, reverted bytes, or a fault. The first terminal result skips the rest of the statement list. Conjunction and disjunction short-circuit; a panic in a required condition propagates before either branch executes. Assignment evaluates its right-hand side first, then checks static mode, then writes the declared field. Invalid declaration indices fault in the evaluator and are rejected by source admission. Falling through a word-returning body returns the zero word; a void body returns empty bytes.

X. Storage layout and initialization

`StorageLayout.lean` derives layout directly from declaration types. The cursor is a slot number and a byte position in $\{0, \ldots, 31\}$. Boolean fields consume one byte from the low-order end. A word field occupies an entire slot; if the cursor is partway through a slot it advances to a fresh one [15]. For example, declarations `bool a; bool b; uint256 c;` occupy $(0, 0), (0, 1), (1, 0)$ respectively.

Let $m_j = 255 \ll 8j$. Boolean extraction tests whether the selected byte is nonzero and returns a canonical zero/one word. Writing Boolean b uses

$$\mathrm{insert}(w, j, b) = (w \,\&\, {\sim}m_j) \mid (\mathrm{encode}(b) \ll 8j).$$

The theorem `PackedBool.insert_preserves_other_bits` preserves every bit outside that byte, including arbitrary unused bits. Whole storage is a function $\mathbb{W} \to \mathbb{W}$, so certificate statements can quantify over arbitrary initial stores. The source parser's layout domain is private word/Boolean declarations; mappings, arrays, inheritance, custom layout, and synthesized public getters require broader source models.

`StorageInitialization.lean` evaluates declaration initializers in order, starting with zero storage. Each word initializer uses checked arithmetic and sees preceding writes; a forward read sees the field's current (default-zero) value. Boolean declaration initializers are currently rejected by the source parser even though the model has Boolean storage operations.

`StorageConstructor.lean` composes these initializers with an optional parameterless constructor body. Nonpayable constructors reject incoming value. The body has its own checked flag, but initializers remain checked even if the body is unchecked. A bare return completes initialization with the current storage, while a source revert or fault preserves its failure classification.

XI. Target semantics and gas

A. Partial raw-byte execution

`Subset.lean` defines a bytecode machine with program counter, word stack, byte memory and allocated-memory word count. Instruction decoding reads raw bytes; jump destinations exclude PUSH data. Stack size is bounded by 1024. Missing code decodes as STOP. The admitted read-only opcodes include word arithmetic, comparisons, bitwise operations, shifts, stack operations, calldata/value access, code copy, memory operations, jumps and terminal instructions. Other opcodes return an explicit unsupported result.

The memory expansion charge for w words is

$$C_{\mathrm{mem}}(w) = 3w + \lfloor w^2/512 \rfloor.$$

Execution has separate outcomes for a completed trace, fault, unsupported opcode and exhausted fuel. Interpreter fuel

bounds the trace length inspected by the proof, while gas accounting applies the target's execution charges. `Execution.lean` proves that a completed unmetered execution with cost c determines execution at every gas budget:

$$\mathrm{meter}(c, o, g) = \begin{cases} \mathrm{finished}(g - c, o) & c \le g, \\ \mathrm{exceptional} & g < c. \end{cases}$$

Larger step bounds preserve the completed result. Gas costs belong to each specific target artifact and can differ across optimized implementations.

B. Storage execution

`Storage.lean` extends the same decoder with SLOAD and SSTORE. Its state contains current storage, warm keys and a signed refund counter; its environment records original storage and static mode. With original, current and new values o, c, n, the SSTORE charge is

$$(\text{warm? } 0 : 2100) + \begin{cases} 100 & c = n, \\ 20000 & o = c = 0,\ c \ne n, \\ 2900 & o = c \ne 0,\ c \ne n, \\ 100 & o \ne c,\ c \ne n. \end{cases}$$

The model applies the Cancun-era warm/cold schedule and refund adjustments; transaction refund settlement and its cap are outside the execution frame. The relevant rule references are EIP-2200, EIP-2929 and EIP-3529 [17], [18], [19].

SSTORE requires remaining gas strictly above 2300 even when consumption is less. Consequently, `Guarded.lean` records both gas used u and minimum initial gas needed h. Charging a step of cost c and guard q to a suffix (u, h) yields

$$(c + u,\ \max(q, c + h)).$$

The metering theorem tests $h \le g$ and returns $g - u$; testing only total consumption would be insufficient. Static SSTORE faults. Frame observation commits successful storage, warming and refunds, while revert/exception retain the pre-frame persistent world and discard reverted effects.

C. Call and creation frames

`Frames.lean` and `StorageFrames.lean` connect machine results to account worlds. A successful call commits incoming value and permitted state changes. A revert retains its bytes and unused gas but rolls back value/state; an exception has empty bytes and no remaining gas. Frame certificates require completed, supported execution.

Creation certificates bind exact creation and runtime bytes, execute initialization and install the certified runtime. They account for execution and code-deposit gas, runtime size and prefix checks, and the initcode size bound [20], [21], [22]. They then derive runtime-call preservation at the deployed account. Their gas budget begins after transaction admission and intrinsic/initcode charges. Address selection, creator nonce admission, collision checks, and transaction fees belong to the external transaction boundary. The interaction model separately represents additional creation operations.

XII. What the certificates prove

Let S be an admitted source model, b exact runtime bytes, k a proved step bound and $i = (d, v)$ an input. The pure-function interface has the shape

$$\begin{aligned} &\mathrm{Preserves}(S, b, k) \implies \\ &\quad \forall i,\ |d| < 2^{64} \implies \exists c, \\ &\qquad \mathrm{run}(b, i, k) = \mathrm{finished}(c, \mathrm{outcome}_S(i)), \end{aligned}$$

together with an inhabited canonical-call domain, the metering result for every gas budget and larger fuel, and preservation of the account world under an installed-code premise. The calldata size bound restricts the certificate's proof domain independently of EVM consensus rules. Domain inhabitation concerns admitted inputs and therefore also covers functions that always panic.

The storage endpoint additionally quantifies over original/current storage, warmth, refunds, static mode and every gas budget. Its source outcome fixes return/revert bytes and complete committed storage; the exact target trace determines execution costs, access effects and refunds. Contract endpoints compose all dispatch paths from bytecode entry. Deployment composition discharges the installed-code premise by deriving the installed image from exact creation execution. Stateful creation requires a successful deployment witness conditional on successful source initialization, allowing always-reverting constructors within the domain.

These implications apply to source/artifact pairs accepted with a complete proof. A sound validator may reject correct compiler output or stop because of unsupported instructions or resource limits. Compiler-specific lowering and artifact capture are external clients responsible for producing the pairs and discharging the certificate premises.

XIII. Trust and exclusions

Certificate validity depends on four explicit boundaries:

1) *Source admission.* Certificate clients must relate complete source text to the formal models. Token recognition, name/type resolution, pragma admission and source-to-model correspondence currently rely on an external, unverified frontend.
2) *Artifact binding.* A certificate client binds exact source names/content, settings, creation bytes and runtime bytes. Completeness and fidelity of the native artifact capture remain obligations of that client.
3) *Model adequacy.* The Lean evaluators must faithfully describe the intended Solidity fragment and Cancun behavior. Agreement with external specifications is supported by source review and differential evidence; a complete mechanized correspondence remains open.
4) *Proof checking.* The pinned Lean kernel checks proof terms. The selected theorem contracts record permitted uses of `propext`, `Classical.choice` and `Quot.sound`. The release combines fresh imported-kernel replay with a

separate manifest comparison. Issuing an artifact certificate additionally requires a proof for the particular source/artifact pair.

The restricted source language of the certificate models excludes general signed/narrow arithmetic, division and remainder expressions, loops, recursion, external calls, inheritance, modifiers, mappings, dynamic ABI data and arbitrary assembly; assembly support is confined to specific raw-return forms. These restrictions apply to the certificate endpoints even where the resolved machine or interaction model supports broader operations. External admission tools may impose further resource limits as implementation policies.

The interaction evaluator uses an explicit call oracle and a creation executor. Its environmental reasoning is conditional on their contracts; verifying the called bytecode and connecting it to those contracts remains a client obligation.

XIV. Resolved scalar semantics and inference

The resolved scalar interface retains source distinctions erased by machine words: payable addresses, enum identity and cardinality, value-type identity and underlying elementary type, contract ancestry, and function visibility, mutability and argument/result types. A resolved declaration registry has an independent well-formedness judgment and an equivalent executable checker. Registry construction and declaration resolution remain later obligations. Aggregate, tuple, metatype and fixed-point forms are represented explicitly. The aggregate operations below consume the resolved registry; tuple effects, metatype members and fixed-point execution retain their separate boundaries.

Independent conversion judgments separate implicit conversion, explicit source cast syntax and internal machine coercion. Integers widen exactly and narrow modulo the target width; fixed bytes resize on the opposite end. Literal conversion requires representability. An out-of-range runtime enum conversion panics with code 0x21, whereas an out-of-range literal conversion rejects statically. Function arguments and results are invariant under implicit conversion, with the specified mutability weakening. Internal function handles may identify compiler-equivalent declarations, so the model permits shared code identities.

The unannotated expression tree records names, literal spellings and source operators. Inference selects operand types and inserts permitted conversions. For example, an expression `x + 256` with `uint8 x` obtains `uint16`; `x + 1` retains `uint8`. Both conditional arms and logical operands are checked before lazy execution. Thus an invalid cast in an unreachable arm still rejects. Checker soundness/completeness is relative to the independent resolved typing judgments over these expression trees.

Binary exponentiation shares the core's width-indexed words. Its wrapping algorithm agrees with mathematical power modulo the word modulus; a saturated integer algorithm implements the checked bounds without constructing a huge power. Universal signed/unsigned laws and correspondence to the independent target power model are proved. Large shift counts use guarded scalar shifts. The pure execution relation permits independent ordinary operand orders; the strategy realizes that relation. A progress theorem excludes static-rejection and unsupported markers from typed expression execution: the result is a value or a Solidity panic. A target-resource claim additionally requires artifact refinement.

Literal semantics covers denominations, scientific notation, numeric hex, address checksums and ASCII/Unicode/hex strings. String UTF-8 checks follow adjacent-token concatenation. Pinned upstream quirks are explicit: a nonzero `1e+2` rejects, `0e+2` is zero, and a leading-zero exponent is parsed as octal, making `1e010` equal to 10^8. Mantissas are normalized before the scientific precision precheck. `SCALARS.md` describes these version-specific compatibility rules. The exact-integer reification of the binary64 precheck and concrete Keccak implementation retain their stated model-adequacy obligations.

Scalar assignment/update plans and pure free-function operator signatures guide location, effect and declaration checking. The continuation machine executes the corresponding lvalue effects and function bodies, while global overload resolution, full parsing, and source-byte binding remain frontend obligations. `SCALARS.md` indexes the scalar interfaces and theorem families.

XV. Aggregate locations and representations

The aggregate interfaces reuse the shared interaction world, byte memory, width-indexed values and concrete Keccak implementation. Persistent and transient account stores are distinct. Arrays, structs, mappings, bytes, strings and calldata slices carry physical references with invocation ownership, packed byte positions, slice descriptors and captured memory binding cells. Reference values clear the former binding cell when read as rvalues.

Layout computes packing, static extents, inherited declaration order and custom persistent bases, with a separate transient cursor. The checked declaration endpoint validates every physical placement, preserves the complete retained declaration inventory and proves bounded cursors and fitting packed fields. The linearization and resolved registry are inputs from declaration elaboration. Runtime storage strides and addresses wrap as 256-bit words. The representation relations allow distinct logical paths to share physical locations, including collisions in Keccak-derived addresses.

Local storage and calldata references alias; memory-to-memory reference assignment also aliases and rebinds a captured member cell. Storage-target copies execute ordered physical operations, including resize-before-copy, buffered packed-word reads and byte cleanup/header order. Independent recursive judgments characterize successful execution, and exact final-store folds retain collisions. Produced scalar observations compose across later disjoint bit fields as well as separate slots. For overlapping copies, the final-store folds account for reads that can observe earlier writes in the same copy. Fresh-memory copies collect immutable-source packets with source-specific decoding/allocation check order and populate new objects. Constructor induction proves scalar words, pointer

cells, headers and payloads, while staged constructor theorems derive freshness from the intermediate allocation frontier.

Defaults permit the canonical shared zero-slot object for empty dynamic values; fixed arrays and structs are recursively initialized. Whole aggregate deletion has a zero-store representation theorem, while mapping payloads retain their physical frame conditions. Scalar deletion preserves packed neighbors. Source guards distinguish proved element removal/layout relocation from an unknown lifetime history invalidated by raw writes or world replacement, which yields an inconclusive observation.

Explicit reference casts retain the resolved location, aliases and slice descriptors. Bytes-to-fixed-bytes conversion reads the used prefix and pads on the right, excluding dirty unused suffixes. Source calldata-tail checks retain native signed/wrapping behavior and distinguish semantic reverts from raw views outside the physical typed domain. ABI copy decoding has its own check order. Host fuel/materialization exhaustion is separate from source panic/revert and from EVM resource adequacy.

The memory relation constrains used packets; unused byte padding and unallocated fringe remain unconstrained. Padding-insensitivity theorems admit native dirty storage-copy packets as well as the canonical interpreter representative. Applying these theorems to compiler-emitted packets requires establishing the representation relation for their concrete bytes.

The implementation separates executable copying from its proof clients. `Operations.lean` implements allocation, conversion, aliasing and physical stores. Its `Laws.lean` companion proves local operation equations, and `Frames.lean` proves world and invocation frames. Memory representation is split into predicates and observations (`Base.lean`), scalar and pointer-cell stores (`Cells.lean`), and constructor correctness (`Construction.lean`), all under the aggregate directory. The `Copy.lean` and `Representation.lean` modules re-export these interfaces. Executable operations and proof clients share one heap and copy interpreter.

XVI. Related work and comparison method

We compare the developments in Table II along four dimensions: executable domain, effect-order and alias representation, proved consumer, and validation oracle. The comparison uses primary papers and, where available, selected interfaces from the cited implementation revisions. Evaluation figures for other projects are taken from their authors' reports.

Hajdu and Jovanović encode local storage pointers and memory allocation/copying for solver automation [7]; da Silva et al. study object identity and proof-oriented data-area representations [8]. The inspected SolidiKeY Lean compiler theorem relates a source fragment to a word-addressed machine under explicit certificate and input-relation premises [9]. Our representation relations track byte-addressed heap packets and ordered physical writes, connecting them to guarded Cancun traces in the pilots. Thornton and Marmsoler prove type safety for Isabelle/Solidity, including consistency of pointer structures [5]. In our continuation machine, the safety invariant connects physical heap representation to captured targets, saved continuations, and the invocation checkpoint across child choices.

TABLE II
Positive claims established by inspected primary artifacts.

Development	Relevant semantic or proof interface
K Solidity [2]	Executable structural operational source semantics in K; contract analysis clients.
Lolisa [3]	Solidity subset in Coq with stronger typed syntax and a reusable formal memory framework.
Isabelle/ Solidity [4], [5], [6]	Deep executable semantics, grammar fuzzing, program proofs and type safety; a separate shallow embedding with a storage model and verification-condition generation.
Solc-verify [7]	SMT-oriented encoding of storage pointers, memory references, allocation and copying.
SolidiKeY [8], [9]	Logical data-area formalization; Lean fragment compilation into a word-addressed machine under explicit input relations. Physical EVM layout and fork/gas correspondence are separate.
SolidCore [10]	Lean source execution with captured storage references and documented intrinsic compiler-specific child orders; reference-toolchain replay.
FORYU [11]	Rocq Yul semantics with a proved liveness-analysis checker and reported evaluation on more than 1,500 contracts.
Solidity Spec	Per-group child choices, physical packet/alias conditions, checked resolved entries and separate source/model/resource observations; explicitly scoped safety and target-proof obligations.

SolidCore is especially close in language and tooling. Its inspected order design fixes construct-specific choices for a compiler configuration. Our machine exposes independent choices for each eager group; its safety theorem must therefore preserve the invariant under every permitted choice in the profile. Isabelle's conformance methodology and FORYU's semantics-plus-checked-consumer evaluation inform our evaluation design [4], [11]. The remaining research question is whether our combined interfaces reduce the assumptions and proof effort of compiler-preservation compositions. Evaluating complete compiler passes would extend the evidence beyond the two bounded pilots presented here.

XVII. Package and reproducibility

The package contains `Contract/` and `Solidity/`, a Lake configuration, and pinned toolchain and dependency files. The default libraries are `Contract` and `SolidparticSemantics`; the only library dependency is Lean's bundled `Std`. The release checks also require Python 3's standard library. The pinned compiler and Anvil are additional dependencies for external conformance checks.

From a Solidity Spec checkout with Lean 4.29.0 installed:

```
make replay JOBS=4
make paper
```

The first command builds the package, runs the audit's negative controls, enforces the theorem manifest and definition-source binding, then freshly rechecks the library and three pilot roots plus scalar regressions sequentially with `leanchecker -fresh`. The second command uses Tectonic to rebuild this paper. CI runs the build and fresh replay and retains diagnostic logs. `RELEASE.md` documents these commands, and `THEOREMS.md` explains the theorem manifest. For external checks, set `SOLC` to the pinned compiler executable and `LEAN_BIN` to the Lean binary directory:

```
python3 scripts/check_conformance.py \
  --solc "$SOLC" --lean-bin "$LEAN_BIN" \
  --output /tmp/new-conformance-run
```

`CONFORMANCE.md` documents observation scope and the required tools.

The public runtime interface is `Solidity.Execution`. Its `Proofs` and `Safety` submodules expose laws and admitted-configuration safety. `Solidity.Compatibility.Execution` exports certificate correspondences. These interfaces separate runtime operations, representation assumptions and proof consumers.

XVIII. Limitations and future research

This release is an incomplete formalization. Resolved declarations, source translation and exact artifact binding are explicit assumptions. The safety profile is narrower than the executable aggregate domain, and the target interpreter covers a Cancun subset. The pilots relate prepared source entries to manually paired bytecode fragments; verified compiler emission and general lowering remain open. Compatibility testing covers selected IR cases, with legacy elaboration left for future work. Generic terminal typing remains existential, and the uniform host bound is far larger than practical executions usually require.

Future research includes verified source and declaration elaboration, safety for additional aggregate representations, linked external execution, and broader ABI and assembly semantics. Compiler-preservation compositions require an explicit observation and gas model and a justified connection to emitted bytes. Further work includes a root-result-indexed terminal invariant, state-dependent resource bounds, verified conversion placement, broader independent execution coverage and proofs for emitted compiler artifacts.

Appendix A
Artifact map

Short filenames in the text identify declarations in their semantic layer. Table III locates those layers. The theorem manifest records fully qualified names and defining modules. `SAFETY.md`, `PILOT.md`, `THEOREMS.md` and `CONFORMANCE.md` index the precise proof profiles, artifact bindings, audit commands and external checks.

TABLE III
Representability and current certificate coverage are distinct.

Layer	Executable meaning and boundary
Typed contract	Indexed variables and slots; finite helper bodies and logs. Parsing, ABI and gas are handled externally.
Certificate source	Resolved word expressions; restricted methods, ABI guards, word/Boolean storage, branches and initialization. Closed parser admission is trusted.
Resolved machine	Physical aggregates, effects, loops and internal recursion. Resolved typed input; safety covers the stated structural profile.
Yul expressions	Word operators and environments used by optimizer proofs; restricted to the expression fragment.
Interaction	Memory/world operations, call oracle, creation executor and fuel, with conditional reasoning about callees.
Bytecode	Partial raw-byte Cancun semantics. Unsupported instructions and exhausted interpreter fuel are explicit non-success results.

References

[1] OKcontract, "Solidity Spec," standalone project. https://github.com/okcontract/solidity-spec.

[2] J. Jiao et al., "Executable Operational Semantics of Solidity," 2018. https://arxiv.org/abs/1804.01295v1.

[3] Z. Yang and H. Lei, "Lolisa: Formal syntax and semantics for a subset of the Solidity programming language in Mathematical Tool Coq," version 3, 2021. https://arxiv.org/abs/1803.09885v3.

[4] D. Marmsoler and A. D. Brucker, "Isabelle/Solidity: A deep embedding of Solidity in Isabelle/HOL," *Formal Aspects of Computing*, 37(2), 2025. https://doi.org/10.1145/3700601.

[5] B. Thornton and D. Marmsoler, "Type Safety for Isabelle/Solidity," author manuscript. https://marmsoler.com/assets/pdf/thornton_24_typesafety.pdf.

[6] D. Marmsoler, A. Ahmed and A. D. Brucker, "Isabelle/Solidity—A shallow Embedding of Solidity in Isabelle/HOL," *Archive of Formal Proofs*, 2026. https://isa-afp.org/entries/Isabelle-Solidity.html.

[7] Á. Hajdu and D. Jovanović, "SMT-Friendly Formalization of the Solidity Memory Model," ESOP 2020, pp. 224–250. https://arxiv.org/abs/2001.03256v2.

[8] G. H. A. da Silva, W. Ahrendt and R. Bubel, "Axiomatisation of Solidity Memory and Storage," SEFM 2025, LNCS 16192, pp. 73–90, 2026. https://doi.org/10.1007/978-3-032-10444-1_5.

[9] SolidiKeY authors, "Solidity Lean," revision `23ba92a3efbc73dce04ad0483294d42757375a0d`, inspected 11 November 2026. https://github.com/SolidiKeY/solidity-lean/tree/23ba92a3efbc73dce04ad0483294d42757375a0d.

[10] Paradigm authors, "SolidCore," revision `1f6fe8636d023926792faf4bd024bba943492a63`, inspected 11 November 2026. https://github.com/paradigmxyz/solidity-lean/tree/1f6fe8636d023926792faf4bd024bba943492a63.

[11] E. Albert, S. Genaim and E. Martin-Martin, "Towards Formally Verified Smart Contracts Compilation," FM 2026, LNCS 16557, pp. 255–274. https://doi.org/10.1007/978-3-032-26220-2_13.

[12] Solidity authors, "Expressions and Control Structures," Solidity 0.8.36 documentation. https://docs.soliditylang.org/en/v0.8.36/control-structures.html.

[13] Solidity authors, "Solidity IR-based Codegen Changes," Solidity 0.8.36 documentation. https://docs.soliditylang.org/en/v0.8.36/ir-breaking-changes.html.

[14] Solidity authors, "Contract ABI Specification," Solidity 0.8.36. https://docs.soliditylang.org/en/v0.8.36/abi-spec.html.

[15] Solidity authors, “Layout of State Variables in Storage and Transient Storage,” Solidity 0.8.36. https://docs.soliditylang.org/en/v0.8.36/internals/layout_in_storage.html.
[16] Solidity authors, “Types,” Solidity 0.8.36. https://docs.soliditylang.org/en/v0.8.36/types.html.
[17] W. Tang, “EIP-2200: Structured Definitions for Net Gas Metering.” https://eips.ethereum.org/EIPS/eip-2200.
[18] V. Buterin and M. Swende, “EIP-2929: Gas cost increases for state access opcodes.” https://eips.ethereum.org/EIPS/eip-2929.
[19] V. Buterin and M. Swende, “EIP-3529: Reduction in refunds.” https://eips.ethereum.org/EIPS/eip-3529.
[20] V. Buterin, “EIP-170: Contract code size limit.” https://eips.ethereum.org/EIPS/eip-170.
[21] A. Beregszaszi et al., “EIP-3541: Reject new contract code starting with the 0xEF byte.” https://eips.ethereum.org/EIPS/eip-3541.
[22] M. H. Swende et al., “EIP-3860: Limit and meter initcode.” https://eips.ethereum.org/EIPS/eip-3860.