Op: A Typed Bytecode for Compliance-Carrying Operations

Raeez Lorgat

April 2026

Op: A Typed Bytecode for Compliance-Carrying Operations

Author: Raeez Lorgat

Abstract

Op is a typed bytecode whose grammar makes typed effects, sanctions dominance, local compensation, typed suspension, linear-and-locked resources, and content-addressed replay primitive rather than library convention. Determinism plus proof bundles are the target condition under which cross-zone verification of institutional operations reduces to pairwise comparison rather than global consensus over compliance state. An Op program is a directed acyclic graph of typed steps with explicit effect rows, precondition/post-condition contracts, scoped compensation attached to the step it inverts, and typed suspension-resumption for callback events. Evaluation is specified to be deterministic, metered on two axes (structural gas over AST nodes, extensional gas over runtime-observed cardinalities), and to emit a content-addressed proof bundle reproducible from the same program, inputs, pack digest, oracle log, and executing-zone identity metadata. The full bit-for-bit replay theorem depends on the canonical-encoding, gas, oracle-log, and terminal-behavior obligations listed in §14. The linear-resource discipline of Move carries over as `Linear<T>` and extends to indexed corridor typestates `Locked<T, ω, ε>`, `Signed<V, ω, ε>`, and `Verified<ω, ε>` that realize the bilateral signed commitment invariants (I_1)-(I_3) of the companion kernel-network paper; the cross-zone signed commitment protocol carries a binary (two-endpoint) session type whose endpoint projections carry a compliance-tensor payload in a distributive lattice. N-ary generalization is future work. Against EVM, WebAssembly, Michelson, and Solidity IR (general-purpose bytecodes whose compliance semantics live in library conventions), Op lifts typed effects, explicit suspension, compensation locality, sanctions-bottom, and jurisdiction scope into grammar. The paper states five conservation invariants or

invariant schemas (gas, linearity, ownership, audit monotonicity, meet-monotonicity) and reports a scoped Qed-closed verdict-preservation result for the admissible Lex-to-Op skeleton; together these specify why cross-zone replay can be pairwise verification instead of consensus when the open obligations in §14 are discharged.

---

1. Workflow as Language, Not Configuration

An institutional workflow is already a program. It composes named steps into a dependency graph, binds step outputs into later-step inputs, branches on runtime conditions, suspends on external events, and reverses partial progress when something fails. Every large institution runs these programs. Almost none of them run them in a programming language.

The programs run in YAML, JSON, visual orchestrators, BPMN diagrams, and ad hoc engines. In those media, step composition is a `depends_on` field. Variable binding is a string interpolation like `"{steps.entity_create.result.id}"`. Branching is an embedded expression fragment that the engine re-parses at runtime. Suspension is a callback token that has no static relationship to the step that produced it. Compensation is a "rollback handler" attached to a workflow scope, detached from the forward step it is meant to invert. Failure policy is an enumeration whose members encode language semantics: `continue`, `halt`, `retry_with_backoff`. The engine implements the semantics; the workflow author discovers them empirically.

None of this is a gratuitous complaint. Workflow engines have their virtues: operators can read the YAML, generators can emit it, the deployment model is simple. But the deficiencies of the medium are structural, and they are load-bearing failure modes in regulated institutions.

The sanctions check skipped in a branch. A workflow writer forgets a sanctions screening gate on a rarely-taken branch. The workflow runs. The sanction-blocked entity opens a treasury account. The error is detected in post-flight audit, weeks later, after the account has moved money.

The registry filing that was never reversed. A workflow partially succeeds, then fails. The rollback handler runs the inverse of the last step and omits the step before that. The registry now has a filing that corresponds to no state in the upstream system. Reconciliation is manual, slow, and expensive.

The uncoordinated cross-zone commit. A cross-zone transfer reaches the second leg. The first leg has already committed. The second leg's kernel replies with an abort. The first leg has no idempotent undo; the ownership change is irreversible in the source-zone's book of record. The two zones are now inconsistent.

The ambiguous writer. Two steps both mutate the same entity record. A race condition, or a human accident, commits the wrong step last. There is no compile-time evidence of which step authored a given row; auditors reconstruct it from timestamps and prayer.

Op replaces the YAML engine with a language. In the specified verifier, a program that type-checks has a typed effect row on every step, a syntactically-attached compensation branch on every step with a compensable effect, a linear-resource discipline that rejects double consumption of a single-use artifact, a typestate discipline that forces exactly one of `commit` or `release` on a cross-zone lock, and a session-typed cross-zone protocol whose payload-abstract bilateral skeleton rules out session-layer deadlocks at well-typed endpoints. The compiler does not run first and discover errors at step 14. The compiler refuses to emit the program, and points at the line.

A *compliance-carrying operation* is what institutional infrastructure produces: a named mutation legal under a named set of rules, executed against a named state, producing evidence that the rules held before the mutation, that the mutation happened, and that any derived rules hold after. An entity formation without the attestation that sanctions cleared is not a formation an auditor will accept. A share issuance without proof that the required governance vote passed is not an issuance a court will enforce. A letter of credit without the signed artifacts of applicant verification, structure validation, and advising-bank confirmation is not a letter of credit a bank will honor. Op is the language in which compliance-carrying operations are written.

Relation to the rule logic. The companion paper *Lex: A Logic for Jurisdictional Rules* specifies a dependently-typed logic for compliance rules. Op's relation to Lex is strict: Op programs declare the compliance obligations they incur (a list of domains), Op steps call Lex predicates as part of their pre-flight and post-flight compliance evaluation, and Op does not re-interpret or modify Lex rules. Op owns control flow, data flow, effects, compensation, and suspension. Lex owns predicate semantics, proof discharge, and the lattice meaning of compliance domains. Section 8 formalizes the boundary.

Relation to the kernel. The companion paper *The Sovereign Jurisdiction Network* specifies the write-path architecture of a proof-producing kernel that is the sole writer to its own database. An Op program is executed by the kernel's operation engine; every mutating step passes through the kernel's mutation-firewall gate and sanctions fail-closed evaluation; the proof bundle each execution produces is the bundle the kernel persists in its append-only event store. Op does not replace the kernel; it is the language in which the kernel's mutating operations are written.

What the paper proves. The paper states five conservation invariants or invariant schemas with proof sketches or proof targets: gas conservation (structural gas charged once per AST node, extensional gas charged by metered read or write, never refunded on failure), resource linearity (values of linear type consumed exactly once), ownership conservation (every state-write has exactly one rightful owner), audit monotonicity (every evaluation step extends the proof bundle without erasing prior entries), and meet-monotonicity (cross-zone composition of Op receipts produces a compliance state no more permissive than any contributing zone). It states two soundness properties: the compilation of the admissible fragment of Lex into Op bytecode preserves Lex verdicts in the target execution for nine compilation cases (§8.5), and the WebAssembly backend of §15 preserves Op execution modulo the terminal abstraction stated there (proof obligation stated; mechanization open, §14). The reverse Lex/Op direction, full abstraction,

and compliance-context adequacy are open theorem obligations, not closed results. The five operational-semantics theorems, termination, progress, subject reduction, effect monotonicity, and parallel confluence, are stated with proof sketches over the Op calculus of §6; their mechanization over Op proper is open work listed in §14, and the toy-fragment instantiations already in the companion Coq development (lambda-calculus, nine-expression gasified AST, two-slot counter machine) do not prove the theorems for Op. §8.5 discloses the scope and Axiom/Parameter inventory of the mechanization precisely.

Scope and limits. Op is a bytecode and proof-artifact specification, not an implementation report and not a claim that every external service invoked by a workflow has already been verified. The paper reports no benchmarks, no backend comparison against EVM, no empirical evaluation under production load, and no deployed system; performance measurement is deferred to future work. The paper is precise about the host assumptions it uses (cryptographic validity, transport delivery within the stated model, and primitive-level contracts). Byzantine liveness beyond the stated model, verified service contracts for every external primitive, and succinct confidential proof transport remain open; they are enumerated alongside the other open problems in Section 14.

Contributions. This paper makes twelve technical contributions.

1. It gives a compliance-carrying typing judgment $\Gamma \mid \kappa \vdash \texttt{e} : \texttt{T} \mid \varepsilon \Rightarrow \kappa'$ that threads pre- and post-compliance states through evaluation, and states progress and preservation for that judgment with proof sketches; mechanization over Op proper is open (§8.5, §14).
2. It specifies the full small-step operational semantics of the bytecode.
3. It introduces bilateral signed commitment (BSC) indexed typestates `Locked<T,`ω`,`ε`>`, `Signed<V,`ω`,`ε`>`, and `Verified<`ω`,`ε`>` realizing the sovereign-jurisdiction-network invariants (I1)-(I3); the three invariants and the certificate-core terminal-evidence determinism lemmas are Qed-closed in Rocq over an abstract corridor history (§8.5).
4. It gives a binary (two-endpoint) session type for cross-zone signed commitment with endpoint projection in the Honda-Vasconcelos-Kubo tradition. Deadlock-freedom and session safety are Qed-closed in Rocq for the bilateral message-ordering skeleton over an uninterpreted payload; the finality/abort evidence core is also Qed-closed, while instantiating the session payload with concrete lock/verdict/-finality witnesses and proving receipt-chain delivery remain explicit proof obligations (§14).
5. It adds Hoare-logic and separation-logic rules, together with rely-guarantee reasoning, a modular method for reasoning about concurrent interference, for cross-zone commit.
6. It states an open information-flow-security theorem schema for the core fragment, meaning high-confidentiality inputs should not change low-observable outputs once the observer model, declassification events, and proof-bundle leakage relation are fixed; mechanization is open (§14).
7. It states the Lex-to-Op adequacy obligations, proves the verdict-soundness direction over nine compilation cases in Rocq, and records weak-bisimulation and full-abstraction arguments as open proof obligations (§8.5, §8.6, §9).

8. It adds refinement types with Liquid- and F*-style weakest-precondition generation, meaning predicates on values are discharged by SMT queries at the typing boundary.
9. It generalizes the corridor witnesses into a typestate calculus that strictly contains the BSC-specific states as one instance.
10. It specifies a PCAuth verifier and transport layer for moving authorized verdict material between zones without erasing provenance.
11. It fixes a CBOR-based binary format for Op bytecode and proof-adjacent transport artifacts; the 5-byte canonical wire-format verifier is Qed-closed for round-trip, injectivity, determinism, and six rejection classes (§8.5).
12. It gives a Wasm ahead-of-time backend and states the CompCert-style correctness obligation, that source-to-target behavior is preserved by explicit compilation theorems rather than trusted translation alone; mechanization is future work (§15, §14).

The cross-zone letter-of-credit case study in Section 12 exercises the full stack end to end, and the multi-harbor settlement example shows the same machinery under corridor composition.

---

2. The Five Grammar Primitives

Op lifts five institutional-workflow primitives from library idiom to language grammar. Each primitive is a response to a specific structural failure of the YAML-engine model.

The five are partly orthogonal and partly layered. Typed effect rows (§2.1), scoped compensation (§2.2), typed suspension (§2.3), linear-and-locked resources (§2.4), and binary session-typed cross-zone signed commitment (§2.5) interact at defined seams: sanctions-dominance is a law on effect rows (§2.1); compensation inherits the forward step's effect row as a lower bound (§2.2, §3.3); `Locked<T, ω, ε>` is a typestate refinement of `Linear<T>` (§2.4); the signed-commitment session-type discipline carries indexed `Locked`, `Signed`, and `Verified` witnesses as payload (§2.5). The abstraction is therefore a language built from three independent primitives (typed effects, linear-and-locked resources, binary session types) and two structural laws (sanctions-dominance on the DAG, compensation locality in syntactic scope). We keep the count of five because each enumerated item corresponds to a distinct construct a programmer writes; the layering is visible at the seams above.

2.1 Typed Effect Rows with Sanctions Dominance

Op tracks nine effects on every step signature. Let `Eff` denote the finite alphabet:

- `sovereign_write`: mutates sovereign-owned state.
- `identity_mutation`: modifies identity state.
- `fiscal_transfer`: moves or commits value.

- `sanctions_check`: queries the sanctions backend (distinguished; see below).
- `governance_request`: initiates a governance ceremony.
- `document_generation`: produces a document artifact.
- `external_read`: reads external state.
- `proof_emit`: emits an attestation.
- `await <event>`: suspends on a callback event.

The effect-row algebra. An effect row is a subset $\rho \subseteq$ `Eff`. Under subset order $\rho_1 \sqsubseteq \rho_2$ iff $\rho_1 \subseteq \rho_2$, union $\rho_1 \sqcup \rho_2$ `:=` $\rho_1 \cup \rho_2$, and bottom $\bot$ `:=` $\emptyset$, the structure `(P(Eff), ⊑, ⊔, ⊥)` is a bounded join-semilattice: $\sqsubseteq$ is reflexive and transitive, $\sqcup$ is associative, commutative, idempotent, and the least upper bound in $\sqsubseteq$, and $\bot$ is below every row. Meet as set intersection exists because the carrier is the finite Boolean lattice `2^9`, but Op's composition operators only join rows. The Qed-closed companion theorem is `effect_row_is_bounded_join_semilattice`.

Each step carries a row in its signature `In` $\rightarrow$ `Out !` ρ. Sequential composition `a ; b` takes row ρ`_a` $\sqcup$ ρ`_b`; parallel composition, guarded choice, and scoped compensation take the corresponding join of branch rows. Effect-row subsumption is the subtyping rule: a function of declared row ρ`'` inhabits a context expecting ρ whenever ρ`'` $\sqsubseteq$ ρ. Section 4.10 derives row propagation from primitive signatures and typing rules rather than stipulating it externally.

Sanctions-dominance as a separation lemma. One effect, `sanctions_check`, is structurally distinguished. Let `W := {sovereign_write, identity_mutation, fiscal_transfer}` $\subseteq$ `Eff` be the write class, and let `G_P` denote the control-flow DAG of program `P`. Op admits no recursion and no unbounded iteration (§3.1, §6.14); the composition operators of §4.4 therefore yield an acyclic expression graph. The sanctions-dominance side condition is:

> ***Side condition (SD).*** *For every step `s` in `G_P` whose declared row contains an effect in `W`, and for every source-to-`s` path in `G_P`, there exists on that path an earlier step `s'` whose declared row contains `sanctions_check`.*

SD is a purely syntactic predicate on declared effect rows along every control-flow path. A program that violates SD does not type-check. One narrow exception admits a post-flight check: `create.entity` may begin before sanctions screening because the subject does not yet exist at the time of the check; screening runs against the created entity after the fact, and a non-compliant outcome aborts creation through pre-registered compensation (§6.12).

The effect row records that a check took place; the verdict lattice records what the check returned. `sanctions_check` is not an absorbing element of the Boolean effect-row algebra. Absorption lives one level up, on the compliance verdict lattice, where `SanctionsBlocked` is bottom under meet and is mechanized in `ComplianceContext.v`.

Three further compile-time checks close adjacent holes: a `fiscal_transfer` without a typed money value is rejected; a `compensate` clause attached to a step with no compensable effect is rejected; a `continue` failure policy on a step carrying `sovereign_write` is rejected.

2.2 Scoped Compensation

A `compensate { ... }` clause attaches syntactically to the step it inverts. The word *inverts* is used in the semantic-compensation sense of Garcia-Molina and Salem (1987): a forward step and its compensating action restore a business-level invariant after abort; the full external state may differ. The retraction reading `inverse ∘ forward = id` is an extensional primitive-contract obligation (§14.9), not a bytecode theorem.

The rollback plan is recorded dynamically as a commit-order stack `C` (§6.9), with the forward DAG carrying the static obligations: scope locality and effect-row subsumption. On failure, inverse actions run in commit-reverse order by the E-Comp-Fail / E-Abort-Pop pass. Siblings do not commute by default. LIFO is the canonical rollback trajectory; a discipline admitting parallel or out-of-order inverse execution would require a commutativity premise on sibling inverses, which Op does not assume.

Each inverse has two obligations. First, the declared inverse is idempotent at the primitive-specification layer: `inverse(inverse(`σ`')) = inverse(`σ`')` for every relevant post-state σ', so a partial rollback can be retried. Second, the declared effect row of the inverse must subsume the forward step's write-class effects. The bytecode type checker enforces the second conjunct. The first is discharged per primitive against the Hoare/separation-logic layer (§11), and its extensional service implementation remains the compensation-correctness open problem (§14.9).

Read against the algebraic-effect tradition, `compensate { inverse }` is a restricted handler for the write-class sub-row of the forward step. The analogy is partial: Op's handler is synthesized from syntax, runs off the main trajectory on abort, and folds over the commit stack rather than handling an operation continuation in place. The useful point is structural: compensation is scoped, typed, and replayable, not a detached cleanup script.

A compensation branch may declare `invalidated_domains`. The host revokes attestation records whose domains overlap the declared set when the rollback runs. A registry filing that is reversed also revokes the `corporate` and `licensing` attestations it produced. The revocation is part of the compensation; no secondary workflow is needed.

The combination of compensation attached to the step it inverts, rollback plan recorded dynamically as a commit-order stack, effect-row subsumption checked statically, idempotency discharged at the primitive-specification layer, and attestation invalidation declared in the compensation block eliminates the syntactic failure where rollback code is detached from the forward step it claims to invert. It does not by itself prove that every external side effect has been semantically undone; that remains the extensional primitive-contract obligation of §14.9.

2.3 Typed Suspension via `await ... within`

An `await e within d` step returns `Await<Event, Payload>`. A downstream step that treats the suspension result as if it were the inner `Payload` (reading fields of the payload before the callback completes, or passing the suspension result to a consumer typed on the payload) fails to type-check.

The continuation is serialized into the proof bundle. When the callback arrives, the kernel resumes the program from the serialized continuation. Resumption is as replayable as any other reduction step: the resume token is content-addressed; the post-resumption configuration is a deterministic function of the pre-suspension configuration and the callback payload; a second kernel replaying the program with the same inputs and the same callback produces the same post-resumption state.

Typed suspension closes the failure class where a workflow blocks on an external event, resumes in an inconsistent intermediate state, and the author has no compile-time evidence of which steps are safe to run in that state.

2.4 Linear and Locked Resources

Two type constructors model single-use artifacts and corridor-local lock state; two further typestates refine the signed-verdict and durable-verification phases of the same cross-zone protocol. We write ω for the operation identifier and ε for the corridor epoch, following the companion kernel-network paper.

- `Linear<T>`: a resource that may be consumed at most once. A program that consumes a linear resource twice fails to type-check; a program that never consumes a linear resource that was introduced into scope fails to type-check. The discipline is strict linearity in Girard's sense (1987): no contraction and no weakening on values of linear type. The carrier `T` need not itself be linear; what is linear is the wrapped inhabitant.
- `Locked<T, ω, ε>`: a resource in a locked state for corridor index `(ω, ε)`. The only eliminators are `commit_transfer(witness_foreign_minted)`, which consumes the resource and produces the destination-side effect, and `release_lock(witness_foreign_aborted)`, which restores the resource to `Linear<T>`. Uniqueness of the active lock at a given side and index is not a stipulation on the constructor: it is a premise of the introduction rule T-Lock of §6.11 on the corridor history context κ, and a structural theorem of that rule set. The mechanization in `BSCInvariants.v` proves it as invariant I1: every typing-rule operation on κ preserves the at-most-one-active-lock-per-`(side, ω, ε)` property.
- `Signed<V, ω, ε>`: a signed verdict witness at corridor index `(ω, ε)`. Uniqueness is a property of the history context, not of the witness itself: T-Sign of §6.11 demands `history_of_X(κ, ω, ε) = ∅` at its introduction, and any second signature by the same side at the same `(ω, ε)` fails the emptiness premise. The mechanization proves this as invariant I2.
- `Verified<ω, ε>`: a durable witness that both sides' `Signed` verdicts at `(ω, ε)` are present in the local corridor history κ, which §7.0 keeps in coherence with the proof bundle μ. Commit consumes

`Locked<T, ω, ε>` only through this witness. Conflicting signed payloads at the same `(ω, ε)` do not form a second `Verified`: they form `Blame<Z, ω, ε>`. The mechanization proves this as invariant I3.

`Locked<T, ω, ε>` is the static witness of the lock phase of a cross-zone signed commitment. Either the destination-zone minting witness arrives and the source-zone resource is consumed, or the destination-zone abort or timeout witness arrives and the source-zone resource is restored to its pre-lock state. `Signed<V, ω, ε>` is the static witness that a side has emitted its unique verdict for that operation and epoch, and `Verified<ω, ε>` is the local static witness that both sides' verdicts are durably present at the evaluator constructing it. Cross-zone terminal agreement additionally depends on the finality-certificate obligation stated in the companion kernel-network paper. The type system forbids any intermediate: a `Locked<T, ω, ε>` that is neither committed nor released is not a well-typed terminal state, and a second local signature at the same index is ill-typed.

Categorical structure. The discipline descends from Girard's linear logic (1987) and Wadler's "Linear Types Can Change the World" (1990). `Linear<T>` marks an object of the linear-type category, the category whose morphisms are not closed under contraction or weakening, rather than itself a binary tensor; the multiplicative conjunction $\otimes$ of linear logic is the pairing of two such objects delivered by the step and parallel combinators of §4. The two eliminators of `Locked<T, ω, ε>` give it the shape of an additive conjunction & (internal choice, selected here by which counterparty witness arrives), not an additive disjunction $\oplus$ (external choice by the producer). Under Wadler's propositions-as-sessions correspondence (ICFP 2012), the consumer-side & branch at the lock endpoint is dual at the counterparty to a $\oplus$ branch that emits either the minted-abroad witness or the aborted-abroad witness.

The four typestates organize into an indexed family over the discrete base category `Corr` of corridor indices `(ω, ε)`. Each fiber is a small typestate category on a given carrier. `release_lock` is the index-forgetting projection `Locked<T, ω, ε>` $\to$ `Linear<T>` out of the fiber of `(ω, ε)`. `Verified<ω, ε>` is formed at an index by a pair of `Signed` witnesses whose payloads are consistent with a well-formed joint verdict; `Blame<Z, ω, ε>` is the typed failure witness for a same-signer conflict at the same index. Other failed attempts to form a terminal witness, including missing finality evidence and malformed imported witnesses, must return typed obstructions rather than being forced into either constructor. A total outcome-classification theorem for `Verified`, `Blame`, `FinalityObstruction`, and `MalformedWitness` is an open obligation. Section 6.11 gives the introduction rules as morphism constructors, and the mechanization discharges the three joint invariants under every one of them.

Op adopts `Linear<T>` from Move (Blackshear et al., 2019) without modification and extends it with the indexed `Locked`, `Signed`, `Verified`, and `Blame` refinements for the cross-zone case, which Move does not address. These surface forms are the smallest cases of the more general typestate calculus in Section 2.6.

2.5 Binary Session-Typed Cross-Zone Signed Commitment

A cross-zone operation is typed as a binary (two-endpoint) session between an Initiator and a Responder kernel under the Honda-Vasconcelos-Kubo binary session-type discipline (ESOP 1998). The session supplies deadlock-freedom and session safety as type-level corollaries for the payload-abstract bilateral skeleton; the indexed corridor typestates `Locked<T, ω, ε>`, `Signed<V, ω, ε>`, and `Verified<ω, ε>` lift the lock, verdict, and durable-verification phases of the signed commitment protocol into static invariants. Section 10 formalizes. The paper mechanizes the binary message-ordering skeleton and states the concrete payload instantiation as an obligation. Residual bilateral foundations used in §10 follow Caires and Pfenning (2010) and Wadler (2012). The Honda-Yoshida-Carbone multiparty discipline (POPL 2008) enters §10 as presentation notation only at `N=2`; generalization to N-ary corridors is future work (§14.21), and this paper's mechanization has no `BG_Par` or `BG_Mu` constructor at this stage.

The session-type discipline is the structural answer to the uncoordinated-commit failure. Two well-typed endpoints conforming to their endpoint projections cannot deadlock on the payload-abstract session layer or accumulate orphan messages in that skeleton. Concrete terminal agreement depends on delivery, finality evidence, timeout policy, and payload instantiation. Equivocation may be reified as a typed blame witness after conflicting signed artifacts are imported, while the watcher and governance layers of the companion kernel-network paper decide how to punish it.

2.6 General Typestate Calculus

The `Linear<T>` and `Locked<T>` forms above are the smallest typestate instances Op needs in its core presentation. The underlying calculus is the broader typestate idea of Strom and Yemini (TSE 1986): resources inhabit programmer-declared state families, transitions are typed witnesses between states, and well-typed programs satisfy the transition obligations induced by their control flow.

User-defined state families. Op therefore admits programmer-defined typestate families:

```
typestate S {
  states s1, s2, ...;
  transitions t1 : s_a → s_b, ...
}
```

If `T` is a carrier type and `S` a declared family, then `T[S, s]` denotes a value of carrier `T` in state `s` of family `S`. A transition witness `t : s_a → s_b` consumes `T[S, s_a]` and produces `T[S, s_b]`. The family declaration is therefore part of the program's type-level interface: each Op operation is checked against the transition graph it declares.

Borrow and inspection. Typestate does not imply that every observation consumes the resource. Op admits Rust-style immutable borrows `&_r T[S, s]`, where `r` is a lexical region:

```
Gamma |- x : T[S, s]
-----------------------------------------------
```

```
Gamma |- borrow_r(x) : &_r T[S, s]  and  x retained
```

The borrow does not consume the original value; the original is retained, and the borrow grants read-only inspection of the same state for the lifetime of region `r`. This is how a program can inspect a pending lock, read a signed verdict, or project session metadata without forcing a state transition.

Local typestate inference. For Op's first-order step language, typestate checking is locally decidable. The compiler generates path-sensitive transition constraints from primitive calls, branch joins, compensation edges, and suspension-resumption boundaries, then infers the most general typestate sequence that satisfies all transition obligations in the local control-flow graph. This is the same flow-sensitive style of local inference Foster, Terauchi, and Aiken study for qualifiers (PLDI 2002): annotations remain useful at exported boundaries, but most interior typestate obligations are inferred rather than hand-written.

Gradual typestate. Not every boundary is fully static. Some external systems return only a runtime witness that a resource is in one of several states, or disclose the state only after a validator runs. Op therefore admits gradual typestate via an unknown state marker `?`: `T[S, ?]` is refined to `T[S, s]` by an explicit runtime check whose success or failure is recorded in the proof bundle. Static and dynamic typestate checking therefore coexist in one program, following the Plaid line of typestate-oriented programming due to Aldrich et al. (OOPSLA 2009).

Monotone families. Some state families are monotone in the sense that later states carry more evidence than earlier ones and never revoke already established facts. Following Pilkiewicz and Pottier (TLDI 2011), such a family carries a preorder `⩽_S` and each transition is required to be monotone with respect to that order. Op uses this shape for audit-bearing states: a successful verification can add a new signed witness or a stronger compliance certificate, but it does not erase the earlier witness from the proof bundle.

Session endpoints as typestate families. Session types and typestate are one calculus in Op. A session role is a typestate family whose states are protocol points and whose transitions are message sends or receives:

```
typestate Role {
  states q0, q1, ...;
  transitions !m : q_i → q_j, ?n : q_j → q_k, ...
}
```

The endpoint projection of Section 8 is therefore a special case of typestate projection. A participant holding a session endpoint of role `Role` also holds a value of type `Endpoint[Role, q_i]`; sending or receiving the next message is the state transition.

The existing Op and BSC forms are instances. The built-in surface forms are all instances of the same calculus. `Linear<T>` abbreviates a family with a live state and one consuming transition to a terminal spent state. `Locked<T>` abbreviates a lock family with source state `locked` and two typed eliminators, `commit` and `release`. The BSC heal's witnesses fit the same pattern. Let `omega` denote the operation's obligation context and `epsilon` its accumulated evidence context. Then `Locked<T,`

`omega, epsilon>`, `Signed<V, omega, epsilon>`, and `Verified<omega, epsilon>` are instances `T[Lock_(omega,epsilon), locked]`, `V[Signature_(omega,epsilon), signed]`, and `Unit[Verification_(omega,epsilon), verified]`. The parameters `omega` and `epsilon` index the family; the state name determines which transitions remain admissible.

Compliance as typestate. An operation itself carries a typestate whose transitions are gated by compliance checks:

```
typestate OpState {
  states pending, executing, committed, aborted, compensated;
  transitions start      : pending   → executing,
              commit     : executing → committed,
              abort      : executing → aborted,
              compensate : aborted   → compensated
}
```

The transitions are not unconditional. `start` requires the pre-flight sanctions and contract checks; `commit` requires that the operation's declared domains are still satisfied and that post-flight evidence has been emitted; `abort` records a failed execution; `compensate` requires the compensation branch and any invalidated-domain revocations to complete. The operation's compliance status is therefore a statically tracked state family rather than an external comment on execution.

Multiparty typestate. The single-owner case generalizes to multiple parties. A global operation may expose one typestate family per participant together with a projection from a shared global family; each participant holds either owned values `T[S_i, s_i]` or borrows `&_r T[S_i, s_i]` against its projected state. A global transition is admissible only when the parties' projected local states jointly satisfy the source side of that transition. This strictly generalizes the MPST projection rule: projection no longer assigns only message actions to endpoints, but coordinated typestate obligations to all parties that share the operation. Cross-zone signed commitment is the binary case; regulated multiparty corridors are the general one.

---

3. The Design Constraints

The constraints below are not aesthetic preferences; they are consequences of what a compliance-carrying operation has to be.

3.1 Determinism

An Op execution given the same program and inputs is specified to produce the same output, side-effect trace, and proof bundle on every kernel that shares the program's content-addressed definition, oracle log, and executing-zone metadata. Determinism makes cross-zone replay well-defined: a receiving zone re-executes a sending zone's Op program against the same inputs and checks that the proof bundle matches un-

der the canonical encoding. The full bit-for-bit replay theorem remains conditional on the open obligations in §14. Any non-determinism (iteration order over an unordered container, clock reads, random-number draws, floating-point rounding modes, non-canonical serialization) breaks replay.

Op enforces determinism grammatically, not by convention. The grammar has no construct for wall-clock reads, no construct for random sampling, no floating-point type, and a canonical-bytes serialization rule for every value.

Determinism extends to terminal outcomes. Given the same program, inputs, rule-pack digests, oracle log, executing-zone identity metadata, cardinality certificates, and callback-event sequence, the phase-terminal form at the end of any `-->` reduction is fixed: value, `Paused(E, K_blob, b)`, `Halted(err, mu)`, `SanctionsBlocked(mu)`, `AuthorizationBlocked(mu)`, `OutOfStructuralGas(mu)`, `OutOfCompensationGas(mu)`, or `Timeout(mu)` (§6.14). When the terminal form is `Paused`, the serialized continuation blob, remaining structural-gas balance, and event-budget counter are byte-identical. Cardinality certificates are therefore inputs to determinism: two runs with the same certificate reach the same gas outcome; two runs with different certificates may reach different gas traps, but each run is deterministic relative to its certificate.

3.2 Explicit Suspension

A compliance operation suspends. It suspends for a governance body to convene, for a regulator to return a verdict, for a counterparty to sign a document. These suspensions are part of the computation. A letter of credit exists because a seller can hand documents to an advising bank and receive the assurance that a distant issuing bank will honor a draft drawn against them when those documents are presented. Suspension is half the operation, and it has legal consequences: operations in `Paused`, `Pending`, and `Completed` states are legally distinct.

Op treats suspension as a typed construct, not a synchronization pattern. A step whose output is `Await<E, T>` suspends pending event `E` and, when resumed, produces a value of type `T`. The evaluator does not spin or poll. It serializes the continuation into a content-addressed blob, stores the blob in the proof bundle, and halts with a structured pause token. When the awaited event arrives, the evaluator recovers the continuation from the blob, validates the event payload against the expected type, and resumes. Suspension is as observable, audit-linkable, and cross-zone-portable as any other step.

3.3 Compensation Attached to Forward

A well-designed institutional workflow specifies its inverse. When the registrar-filing step succeeds but the cap-table step fails, the registrar filing must be reversed; otherwise the filed share-capital increase is uncompensated by any corresponding change to the register, and the entity's public record drifts out of sync with its internal state. The inverse is a specific, named action that undoes a specific, named forward action; it is not a general rollback.

In YAML workflow formats, the inverse is authored in a detached compensation block at the end of the file, readable against its forward only by a human who keeps both in mind. Op rejects this: a step and its compensation form a single syntactic unit. The compensation block appears as a clause attached to the step body, sharing the step's scope and inheriting its typed inputs and outputs. The compiler rejects any step whose compensation references a variable the forward step does not bind, and any step whose forward side-effect lacks a declared compensation unless the forward effect is marked non-compensable. Compensation locality is a precondition for the compiler to prove that compensation, when run, inverts the forward; it is not a stylistic preference.

3.4 Sanctions Dominance

Some obligations are fail-closed after applicability is resolved. Sanctions are the paradigm: the International Emergency Economic Powers Act, the UN Consolidated Sanctions List, the EU Consolidated List, and their national-law analogues require the evaluator to account for licenses, humanitarian carveouts, sovereign exemptions, and shared-authority recognition before it returns its verdict. Once the applicable Sanctions coordinate returns `NonCompliant`, no other compliance consideration raises that coordinate. A step that performs a mutation for the benefit of a sanctioned entity is not a step that should succeed, be compensated, and produce a record of the attempt. It is a step that should not begin.

Op encodes this as a language-level law, not a runtime pattern. The effect `sanctions_check` is distinguished: its failure produces the bottom type (a type with no inhabitants), which the evaluator cannot recover from and which propagates through any continuation attempting to use its result. This is the operational counterpart of the sanctions-dominance rule in the companion rule logic. At the grammar level, a step performing `sovereign_write`, `identity_mutation`, or `fiscal_transfer` must be dominated in the DAG by a `sanctions_check` step, with one narrow exception for the creation of an entity that does not yet have an identity to screen. The type checker rejects dominance violations before execution begins.

3.5 Cross-Zone Replay

A compliance-carrying operation may need verification by a jurisdiction other than the one that executed it. This is the normal case for multi-harbor entities: an entity harbored in jurisdictions A and B executes an operation in A, and B wants to verify what happened. B does not re-run A's workflow; B confirms that the workflow A ran was the workflow A's jurisdiction specified, that the inputs A fed to the workflow were the inputs A committed to, and that the workflow, run against those inputs under A's rule pack, produces the output A claimed.

Op supports this by making every execution's inputs, program definition, and output a content-addressed artifact and by committing the execution trace as a hash-chain. The structural discipline descends from proof-carrying code (Necula, POPL 1997): the program ships together with a machine-checkable record of the conditions under which its behaviour is sanctioned. Cross-zone replay asks: "given this program hash, this input hash, this pack version, and this event sequence, does replay on a local evaluator reproduce

the claimed output hash?" A yes answer is as good as having executed the operation locally. A no answer is provable divergence between what A claimed and the program's executed behavior. The architecture does not rely on either party trusting the other's evaluator; it relies on both parties trusting the evaluator's determinism specification, which is the subject of this paper.

3.6 Portability Without Blockchain

The constraint is not "portability implemented via a shared ledger." Cross-zone replay requires no global consensus over compliance state; the Op trace is self-contained. The receiving zone replays against its own evaluator, compares outputs, and accepts or rejects. The alternative, a shared blockchain on which compliance operations are globally ordered, would require jurisdictions to agree on a single ordering authority, the political configuration the sovereign-jurisdiction architecture is designed to avoid. Op inherits the no-global-consensus property from its host architecture; the replay protocol is pairwise verification, not consensus participation.

These six constraints are the brief. The rest of the paper specifies a language that satisfies them.

3.7 Bytecode Binary Format

Content addressing commits to bytes, not to informal AST identity. Op therefore fixes an explicit wire format. The wire format is canonical CBOR as standardized by Bormann and Hoffman, *Concise Binary Object Representation (CBOR)*, RFC 8949 (2020). The digest commitment for Op programs and other content-addressed artifacts is BLAKE3-256 over those canonical bytes; Bertoni, Daemen, Peeters, and Van Assche, *Keccak/SHA-3* (2013), is the standardized comparison point; the pinned Op choice follows Aumasson, Neves, Wilcox-O'Hearn, Winnerlein, *BLAKE3* (2020). A digest computed over any non-canonical encoding is invalid.

Program object. The root CBOR item of an Op program is `tag(60000)` applied to a definite-length map with exactly four required keys:

```
tag(60000)({
  0: version : uint,
  1: declared_types : [TypeDecl],
  2: declared_effects : [EffectDecl],
  3: body : Operator
})
```

Version `1` is the format fixed in this paper. Under version `1` the map is closed: missing keys, duplicate keys, or unknown keys are parse errors. `declared_types` is an ordered array of `tag(60001){0: name, 1: type_term}` declarations, where prelude scalar types are encoded by their stable 16-bit prelude codes and compound types are encoded as tagged applications of the constructors `record`, `variant`, `list`, `option`, `result`, `linear`, and `locked`. `declared_effects` is a duplicate-free array of effect descriptors sorted by their deterministic CBOR encodings. The non-parameterized effects use one-byte prelude

codes `0..7` for `sovereign_write`, `identity_mutation`, `fiscal_transfer`, `sanctions_check`, `governance_request`, `document_generation`, `external_read`, and `proof_emit`; the parameterized effect `await <event>` is `tag(60002)[8, event_type_code]`.

Operators. The program body is a tagged operator tree. Every operator uses a distinct CBOR tag and a fixed operand shape:

```
tag(60010)[type_term, literal_value]                -- literal
tag(60011)identifier_text                           -- variable
tag(60012)[binder, type_term, rhs, body]            -- let
tag(60013)[left, right]                             -- seq
tag(60014)[op_0, ..., op_n]                         -- par, n ⩾ 1
tag(60015)[[guard_0, block_0], ..., [null, else_b]] -- choose
tag(60016)({0:id, 1:in_t, 2:out_t, 3:effects,
            4:body, 5:compensation_or_null,
            6:requires, 7:ensures})                 -- step
tag(60017)[primitive_id, [arg_0, ..., arg_n]]       -- call
tag(60018)[event_type_code, deadline, resume_type]  -- await
tag(60019)[jurisdiction_id, body]                   -- in_jurisdiction
tag(60020)[hole_id, value, pcauth_witness]          -- fill
```

Wrong arity, wrong tag, or wrong operand type is a parse error. Operator arrays are definite-length arrays only. Set-valued operands such as effect rows are normalized to sorted duplicate-free arrays before encoding, so the byte representation does not depend on source ordering.

Compliance contexts and verdicts. A compliance context is `tag(60030)` applied to a definite-length map from standard-prelude domain codes to verdict atoms. Domain identifiers are stable unsigned 16-bit codes:

```
0x0001 aml               0x0009 licensing          0x0011 ip
0x0002 kyc               0x000A banking            0x0012 consumer_protection
0x0003 sanctions         0x000B payments           0x0013 arbitration
0x0004 tax               0x000C clearing           0x0014 trade
0x0005 securities        0x000D settlement         0x0015 insurance
0x0006 corporate         0x000E digital_assets     0x0016 anti_bribery
0x0007 custody           0x000F employment         0x0017 sharia
0x0008 data_privacy      0x0010 immigration
```

Tensor cells encode two axes, not one flat verdict enum. The compliance-grade atoms are the unsigned integers `0 = NonCompliant`, `1 = Pending`, and `2 = Compliant` on the Applicable fragment. `NotApplicable` and `Exempt` are applicability markers carried by the enclosing `TensorValue` variant. `SanctionsBlocked` is not a verdict atom in a compliance context; it is a terminal proof-bundle entry.

Linear and locked typestates. The pair `(omega, epsilon)` is the static typestate index of a linear resource: `omega` names the unique introduction site of the resource and `epsilon` names the unique elimination obligation the type checker tracks for that resource. Runtime values of `Linear<T>` and `Locked<T>` are encoded as:

```
tag(60040)[type_term, omega, epsilon, payload]   -- Linear<T>
tag(60041)[type_term, omega, epsilon, payload]   -- Locked<T>
```

Both indices are encoded as unsigned integers in shortest form. The distinct tags make the typestate observable at the byte level.

PCAuth witnesses. A PCAuth witness is structured CBOR indexed by both the filled value and its type. Write `W_tau(v)` for the witness grammar at type `tau` and value `v`:

```
W_tau(v) =
  tag(60050)({
    0: hole_id,
    1: tau,
    2: canonical_value_bytes,
    3: predicate_digest,
    4: authority_id,
    5: scope_digest,
    6: subwitnesses,
    7: authority_signature
  })
```

For scalar `tau`, `subwitnesses` is `null`. For a record type, `subwitnesses` is a map from field names to the corresponding field witnesses; for a variant, it is the pair `[constructor_tag, payload_witness]`; for a list, it is the ordered array of element witnesses. The field `canonical_value_bytes` is exactly the canonical CBOR encoding of the filled value; the signature covers the canonical CBOR encoding of keys `0..6` under the surrounding `tag(60050)`.

Proof bundles and oracle logs. The proof bundle is an ordered sequence `tag(60060)[entry_0, ..., entry_n]`. Each entry is tagged and carries a strictly increasing ordinal in key `0`. The core entry tags are:

```
tag(60061){0:ord, 1:step_id, 2:input_digest, 3:output_digest,
           4:effects, 5:receipt_or_null}          -- commit
tag(60062){0:ord, 1:event_type, 2:continuation_digest,
           3:deadline}                            -- pause
tag(60063){0:ord, 1:event_digest}                 -- resume
tag(60064){0:ord, 1:step_id, 2:inverse_digest}    -- compensation
tag(60065){0:ord, 1:witness_digest}               -- pcauth_attach
tag(60066){0:ord, 1:oracle_log_index}             -- oracle_attach
tag(60067){0:ord, 1:terminal_code}                -- terminal
```

The oracle log is an ordered sequence `tag(60070)[oracle_0, ..., oracle_n]` whose order is the logical chronology of evaluation, not wall-clock time. Each oracle entry is:

```
tag(60071)({
  0: ordinal,
  1: kind,
  2: request_digest,
  3: response_item,
```

```
  4: attestation_payload,
  5: [tag(60072)[signer_id, sig_alg, signature_bytes], ...]
})
```

The signatures attest the canonical CBOR encoding of keys `0..4` under `tag(60071)`. Out-of-order ordinals, duplicate ordinals, or missing signatures are parse errors.

Canonicalization. Op version `1` adopts the strict deterministic CBOR rules of RFC 8949 §4.2.1. Preferred serialization is mandatory: integers, lengths, and tag numbers use the shortest available form; definite lengths are mandatory; indefinite-length strings, arrays, and maps are forbidden; and every map key is sorted by the bytewise lexicographic order of its deterministic CBOR encoding. Op version `1` adds three application-level restrictions needed for unique program bytes: floating-point values are forbidden; duplicate map keys are rejected; and any set-valued field is encoded as a duplicate-free array sorted by deterministic encoding. The encoder is therefore a total function from well-typed Op ASTs to unique byte strings.

Content addressing and the collision-resistance assumption. Let `enc(P)` be the canonical CBOR encoding of program `P`. Op version `1` defines the content address of `P` as `BLAKE3-256(enc(P))`, a 256-bit digest (32 bytes). The cryptographic assumption is the standard collision-resistance assumption for BLAKE3-256: for any probabilistic polynomial-time adversary, finding distinct byte strings `x ≠ y` such that `BLAKE3-256(x) = BLAKE3-256(y)` is computationally infeasible.

Parser.

```
parse : Bytes → Result Op_AST | ParseError(reason)
```

It first decodes one CBOR item, rejects any non-deterministic encoding, checks the root tag and version, and then recursively validates tags, arities, domain codes, effect codes, and typestate indices. For any finite byte string, `parse` terminates, because every recursive call consumes at least one byte of a definite-length CBOR item. A streaming implementation uses memory bounded by the nesting depth plus the largest currently-open definite-length item, and it must reject any declared size above the implementation cap with `ParseError(ResourceBoundExceeded)`. The distinguished parse errors are `UnsupportedVersion`, `NonCanonical`, `UnknownTag`, `DuplicateKey`, `UnknownDomain`, `UnknownEffect`, `Arity`, and `ResourceBoundExceeded`.

Equivocation defense. Two semantically distinct programs cannot share the same canonical encoding. The proof is direct. Every sum type in the grammar has a distinct CBOR tag; every product type uses either a fixed-position array or a fixed-key map with unique integer keys; every set-valued field is normalized before encoding; and RFC 8949 §4.2.1 removes alternative byte encodings for the same CBOR item. Therefore the canonical encoder `enc` is injective. If `enc(P_1) = enc(P_2)`, then `parse(enc(P_1)) = P_1` and `parse(enc(P_2)) = P_2`, so totality of `parse` yields `P_1 = P_2`. Semantically distinct programs are therefore byte-distinct. At the digest layer, any equality `BLAKE3-256(enc(P_1)) = BLAKE3-256(enc(P_2))` for `P_1 ≠ P_2` would be a collision in BLAKE3-256.

Backwards compatibility. The version field is part of the hashed bytes and is therefore part of the content address. Future format extensions proceed by incrementing `version` and defining new tags or new map keys. A version-`1` parser rejects later versions rather than silently misparsing them; a later parser may admit both version-`1` and version-`k` inputs. Format evolution therefore changes the content address explicitly instead of creating serialization aliases across versions.

---

4. The Op Language

4.1 Programs

An Op program is a named executable workflow. Its identity is a triple (name, jurisdiction, pack-digest) where name is a dotted identifier, jurisdiction is a canonicalized jurisdiction reference, and pack-digest is the content-addressed hash of the rule pack against which the program's compliance obligations were checked at compile time. The program declares typed inputs (a record type), typed outputs (a record type), an effect row, a contract clause (preconditions and postconditions as compliance-domain shorthand or as predicate expressions), and a body (a sequence of statements composed from the operators below).

4.2 Steps

A step is the smallest unit that can mutate or suspend. Every step has:

- A stable identifier (a dotted name).
- An input type.
- An output type.
- An effect row.
- A body (the forward computation).
- An optional compensation clause (the inverse).
- Optional precondition and postcondition assertions.
- Optional suspension semantics.

Step signatures are written:

```
step name : In -> Out ! Effects
```

Steps compose via the operators in 3.4.

4.3 Primitives

A primitive is a kernel-recognized action with a stable lowering rule. Primitives form the leaves of an Op program's expression tree. The primitive taxonomy is kernel-defined and pack-versioned; the taxonomy includes create-operations (entity creation, treasury creation, bank account creation), update-operations

(entity status, cap table), consent-operations (board resolution, member resolution, shareholder vote), screening-operations (sanctions screening), document-operations (board minutes, shareholder minutes, filing artifacts), trade-operations (invoice creation, letter of credit issuance), and filing-operations (registry amendment). Each primitive carries a typed signature, a default effect row, and a lowering rule to a canonical service call. New primitives register in a new pack version without modifying the language.

4.4 Composition Operators

Op provides five composition operators; Section 6 states each one's small-step reduction rule.

Sequential composition. `s ; t` runs step `s` to completion, binds its output into the environment, then runs `t`. Binding is by name: `s` produces a typed value under its step name, and `t` projects fields from that value.

Parallel composition. `par { a = e_a; b = e_b; ... }` runs the expressions against the common incoming environment and collects their results. Parallel branches cannot have mutual data dependencies; the type checker rejects a branch that reads from a sibling's output.

Guarded choice. `choose { when cond_1 → block_1; ... else → block_n }` evaluates the guard expressions in source order; the first `true` selects its block. Blocks must have unified output types and mutually disjoint effect upper bounds.

Suspension. `await e within d` is a term of type `Await<e, T>` where `T` is the payload type associated with callback event `e` in the runtime's callback type registry. An await suspends the computation; the event's arrival resumes it with a value of type `T`.

Scoped compensation. `s { body } compensate { inverse }` attaches an inverse block `inverse` to step `s`'s forward body. The inverse block type-checks in `s`'s scope, with access to `s`'s inputs and to its committed output (if any).

4.5 Jurisdiction Scope

Every Op program has one ambient jurisdiction that authorizes its compliance obligations. The scoping operator `in j { block }` rebinds the ambient jurisdiction for the duration of `block`. Scoped jurisdictions change the interpretation of compliance obligations, the rule pack consulted for predicate evaluation, and the identity of the regulator to whom post-flight evidence is attributed. A program using `in j { ... }` commits to the jurisdictions it scopes over; the compiler rejects a scope rebind that would change a primitive's effect row in a way that violates the sanctions-dominance law (Section 3.4).

4.6 Participant and Approval Declarations

Multi-entity operations (cross-entity transfers, mergers, and settlement between counterparties) require explicit participant declarations. A participant clause names each entity in the operation, assigns a role

(acquirer, target, source-zone, destination-zone, bilateral-counterparty, ordinary-participant), and declares the governance the participant's home jurisdiction requires. An approval clause specifies the composition rule (unanimous, majority, specific, bilateral) under which participant approvals compose into an aggregate approval. Participant declarations are not decoration; they are inputs to the cross-zone commit protocol (Section 10.7) and to the multi-entity tensor composition (the companion algebra paper).

4.7 Contracts

A step or a program may declare:

```
requires domains [...]; ensures domains [...];
```

The `requires` clause lists compliance domains whose Applicable cells must be `Compliant` for the step to begin. `Pending`, `NonCompliant`, missing evidence, and missing corridor translation block ordinary execution. `Exempt` and `NotApplicable` are accepted only as structured applicability outcomes with provenance, not as permissive verdict atoms. The `ensures` clause lists domains for which the step produces evidence on success. Contracts can also be boolean expressions over the input environment, in which case they are first-order predicates checked before and after step execution. Predicate-form contracts are the interface to the rule logic: a compliance predicate is a Lex term, compiled into an Op-visible boolean expression, and attached as a precondition or postcondition. Section 8 formalizes this.

The domain names appearing in a contract clause resolve against the compliance-domain standard library of Section 4.8 together with any pack-defined extensions.

4.8 Compliance-domain standard library

Op fixes an interoperable standard library of 23 compliance-domain modules at the bytecode layer. The standard library remains open: a jurisdiction pack may add further domains or aliases, and every conforming evaluator must assign identical semantics to the 23 modules below. The reserved standard-library code range is `0x0101-0x0703`; pack-defined extensions must use codes outside that range and publish an order-embedding into the Applicable-grade lattice.

Each standard-library domain `d` exports a 16-bit code `c_d`, a version witness `nu_d`, a verdict carrier `V_d`, a trusted oracle type `O_d`, and a sovereign attestation-key root `kappa_d`. Its bytecode-visible state record is:

```
DomainState<code, V> = {
  code: u16,
  version: DomainVersion,
  verdict: V,
  oracle_root: Digest,
  attestation_root: Digest
}
```

The carrier `V_d` embeds monotonically into the Applicable-fragment per-domain grade lattice used elsewhere in the paper. At minimum every Applicable carrier contains `NonCompliant`, `Pending`, and `Compliant`; families may refine the interior with evidence-specific constructors without changing the order. `Exempt` and `NotApplicable` are carried as applicability markers outside this grade lattice.

Let `D_std` be the set of the 23 standard-library codes. A standard-library tensor is a function `T : D_std → TensorValue`. On coordinates where both inputs are Applicable, composition is the pointwise meet on the grade lattice: `(T_A sqcap T_B)(d) = T_A(d) meet T_B(d)`. On mixed applicability coordinates, composition returns a structured `MeetResult` rather than a single lattice value.

Family aliases in contract clauses are themselves meets: if `L(F)` is the leaf-set of a family `F`, then `T(F) = meet_{d in L(F)} T(d)`. Thus `requires domains [sanctions]` requires the meet of `OFAC`, `UN`, `EU`, `ADGM`, and `DIFC`; likewise for `kyc`, `aml`, `tax`, `securities`, `privacy`, and `sharia`.

Sanctions family

- `OFAC` (`0x0101`, version `nu_ofac`): `DomainState<0x0101, SanctionsVerdict>`; predicates `listed`, `blocked_program`, `fifty_percent_owned`; PCAuth `Sanctions.Officer`; oracle `OfacListOracle`; key root `kappa_ofac`.
- `UN` (`0x0102`, version `nu_un`): `DomainState<0x0102, SanctionsVerdict>`; predicates `listed`, `terror_designation`, `arms_embargo_subject`; PCAuth `Sanctions.Officer`; oracle `UNSanctionsOracle`; key root `kappa_un`.
- `EU` (`0x0103`, version `nu_eu`): `DomainState<0x0103, SanctionsVerdict>`; predicates `listed`, `sectoral_restriction`, `asset_freeze_subject`; PCAuth `Sanctions.Officer`; oracle `EUSanctionsOracle`; key root `kappa_eu`.
- `ADGM` (`0x0104`, version `nu_adgm`): `DomainState<0x0104, SanctionsVerdict>`; predicates `listed`, `fsra_watchlist_match`, `adgm_prohibited_counterparty`; PCAuth `Sanctions.Officer`; oracle `ADGMSanctionsOracle`; key root `kappa_adgm`.
- `DIFC` (`0x0105`, version `nu_difc`): `DomainState<0x0105, SanctionsVerdict>`; predicates `listed`, `dfsa_watchlist_match`, `difc_prohibited_counterparty`; PCAuth `Sanctions.Officer`; oracle `DIFCSanctionsOracle`; key root `kappa_difc`.

KYC family

- `identity_verification` (`0x0201`, version `nu_idv`): `DomainState<0x0201, KYCVerdict>`; predicates `legal_name_match`, `document_valid`, `liveness_passed`; PCAuth `KYC.ApprovedPerson`; oracle `IdentityVerificationOracle`; key root `kappa_idv`.
- `source_of_funds` (`0x0202`, version `nu_sof`): `DomainState<0x0202, KYCVerdict>`; predicates `income_traceable`, `provenance_complete`, `adverse_media_clear`; PCAuth `KYC.ApprovedPerson`; oracle `SourceOfFundsOracle`; key root `kappa_sof`.

- `beneficial_ownership` (`0x0203`, version `nu_ubo`): `DomainState<0x0203, KYCVerdict>`; predicates `ubo_chain_complete`, `ubo_threshold_satisfied`, `control_person_identified`; PCAuth `KYC.ApprovedPerson`; oracle `BeneficialOwnershipOracle`; key root `kappa_ubo`.

AML family

- `txn_monitoring` (`0x0301`, version `nu_txn`): `DomainState<0x0301, AMLVerdict>`; predicates `pattern_score_below_threshold`, `counterparty_risk_acceptable`, `velocity_within_profile`; PCAuth `AML.Investigator`; oracle `TransactionMonitoringOracle`; key root `kappa_txn`.
- `structuring_detection` (`0x0302`, version `nu_struct`): `DomainState<0x0302, AMLVerdict>`; predicates `aggregate_below_reporting_split`, `burst_pattern_absent`, `smurfing_cluster_absent`; PCAuth `AML.Investigator`; oracle `StructuringDetectionOracle`; key root `kappa_struct`.
- `large_value_reporting` (`0x0303`, version `nu_lvr`): `DomainState<0x0303, AMLVerdict>`; predicates `report_filed`, `regulator_acknowledged`, `filing_deadline_open`; PCAuth `AML.Investigator`; oracle `LargeValueReportingOracle`; key root `kappa_lvr`.

Tax family

- `FATCA` (`0x0401`, version `nu_fatca`): `DomainState<0x0401, TaxVerdict>`; predicates `us_person_classified`, `fatca_form_collected`, `reportable_account_tagged`; PCAuth `Tax.ReportingOfficer`; oracle `FATCAOracle`; key root `kappa_fatca`.
- `CRS` (`0x0402`, version `nu_crs`): `DomainState<0x0402, TaxVerdict>`; predicates `tax_residency_declared`, `self_certification_valid`, `reportable_jurisdiction_mapped`; PCAuth `Tax.ReportingOfficer`; oracle `CRSOracle`; key root `kappa_crs`.
- `withholding` (`0x0403`, version `nu_withholding`): `DomainState<0x0403, TaxVerdict>`; predicates `treaty_rate_resolved`, `beneficial_owner_verified`, `withholding_applied`; PCAuth `Tax.ReportingOfficer`; oracle `WithholdingOracle`; key root `kappa_withholding`.

Securities family

- `eligibility` (`0x0501`, version `nu_eligibility`): `DomainState<0x0501, SecuritiesVerdict>`; predicates `investor_class_permitted`, `offering_exemption_available`, `issuer_type_permitted`; PCAuth `Securities.Supervisor`; oracle `EligibilityOracle`; key root `kappa_eligibility`.
- `holding_period` (`0x0502`, version `nu_holding`): `DomainState<0x0502, SecuritiesVerdict>`; predicates `lockup_elapsed`, `transfer_window_open`, `affiliate_restriction_cleared`; PCAuth `Securities.Supervisor`; oracle `HoldingPeriodOracle`; key root `kappa_holding`.

- `suitability` (`0x0503`, version `nu_suitability`): `DomainState<0x0503, SecuritiesVerdict>`; predicates `risk_profile_matched`, `knowledge_assessed`, `concentration_within_limit`; PCAuth `Securities.Supervisor`; oracle `SuitabilityOracle`; key root `kappa_suitability`.

Privacy family

- `GDPR_consent` (`0x0601`, version `nu_gdpr`): `DomainState<0x0601, PrivacyVerdict>`; predicates `purpose_specific`, `freely_given`, `withdrawal_available`; PCAuth `Privacy.DataProtectionOfficer`; oracle `GDPRConsentOracle`; key root `kappa_gdpr`.
- `data_residency` (`0x0602`, version `nu_residency`): `DomainState<0x0602, PrivacyVerdict>`; predicates `storage_region_permitted`, `subprocessor_region_permitted`, `backup_region_permitted`; PCAuth `Privacy.DataProtectionOfficer`; oracle `DataResidencyOracle`; key root `kappa_residency`.
- `cross_border_transfer` (`0x0603`, version `nu_cbt`): `DomainState<0x0603, PrivacyVerdict>`; predicates `transfer_mechanism_valid`, `destination_adequate`, `supplementary_measures_present`; PCAuth `Privacy.DataProtectionOfficer`; oracle `CrossBorderTransferOracle`; key root `kappa_cbt`.

Sharia family

- `interest_prohibition` (`0x0701`, version `nu_riba`): `DomainState<0x0701, ShariaVerdict>`; predicates `riba_absent`, `profit_basis_disclosed`, `late_fee_purified`; PCAuth `Sharia.BoardScholar`; oracle `InterestProhibitionOracle`; key root `kappa_riba`.
- `asset_screen` (`0x0702`, version `nu_asset`): `DomainState<0x0702, ShariaVerdict>`; predicates `permissible_asset_class`, `revenue_screen_passed`, `debt_ratio_below_limit`; PCAuth `Sharia.BoardScholar`; oracle `AssetScreenOracle`; key root `kappa_asset`.
- `gharar_threshold` (`0x0703`, version `nu_gharar`): `DomainState<0x0703, ShariaVerdict>`; predicates `material_terms_certain`, `delivery_conditions_specified`, `speculation_below_limit`; PCAuth `Sharia.BoardScholar`; oracle `GhararThresholdOracle`; key root `kappa_gharar`.

4.9 Gas

Every Op program declares a gas budget. Gas is metered on two axes:

- **Structural gas.** One unit per small-step reduction, including reductions entered by resumption from a serialized continuation. The structural-gas bound is the count of reduction-rule-applying forms in the fully compiled AST (after admissible-fragment gating, after `fill` substitution, and after let-inlining of literal records) times the sum of declared maximum resumption counts across every `await` site in the program. The compiler computes the bound on the compiled AST, not the source AST; the source

AST's node count is a strict lower bound under substitutions that expand literal values. The compiler rejects programs with undeclared resumption multiplicity.

- **Extensional gas.** Consumed by operations whose cost depends on runtime-observed quantities: storage growth, payload size, number of cross-zone receipts consulted, number of domains re-evaluated. Each contributing operation has a published rate (units per byte, per receipt, per domain). Extensional gas admits static bounds only when cardinalities are known at compile time; otherwise the compiler emits a cardinality obligation: a runtime certificate that the cardinalities are within stated limits, checked at dispatch.

A cardinality certificate is a typed object `Cert = {dimension_id : CardinalityDim; upper_bound : Nat; issuer : PrincipalId; signature : Sig}`, where `CardinalityDim` enumerates extensional-gas dimensions such as payload bytes, receipts consulted, domains re-evaluated, and storage-growth bytes. The signature binds the dimension, bound, and issuer. At dispatch of a metered primitive, the reducer verifies the signature, fetches the published rate for the dimension, requires that `G.extensional` can pay the declared upper bound, and records any breached bound in `mu` if the runtime value exceeds the certificate. Certificates are dynamic inputs to the operational semantics, not static typing constraints.

Gas is charged simultaneously with the reduction that performs the work: the small-step rule's conclusion writes the post-reduction balance, so the charge and the reduction are one atomic transition. The pre-charge intuition is operationalized by terminal traps: a rule whose pre-state lacks sufficient structural or extensional budget cannot fire normally and instead fails closed, runs compensation if declared, and records the failure in the proof bundle. Gas is never refunded on failure; the cost of a failed attempt is paid.

The declared gas budget `G_total` covers both the forward path and any compensation path executed on abort. Compensation runs charge against the remaining balance of `G_total` after the failure, not against a fresh allowance; a program whose forward path exhausts `G_total` at step `N` and whose compensation for steps `1..N-1` cannot complete within the remaining balance halts with `OutOfCompensationGas` in the proof bundle, leaving residual forward commits for operator reconciliation. The compiler requires every program to declare a separate `compensation_budget` sub-allocation of `G_total`; programs whose declared compensation budget is insufficient for the worst-case compensation path (the sum of per-step compensation rates over the maximum possible set of committed forward steps) are rejected.

Section 7.1 states the gas-conservation invariant.

4.10 Effects

Op tracks a fixed effect vocabulary. The effects are:

- `sovereign_write`: a mutation of kernel-owned sovereign state.
- `identity_mutation`: a mutation of identity-subsystem state.

- `fiscal_transfer`: a value movement or commitment.
- `sanctions_check`: a screening against sanctions lists; distinguished because its failure produces the bottom type.
- `governance_request`: a request to a governance body.
- `document_generation`: the production of a signed artifact.
- `external_read`: a read of external state through an oracle.
- `proof_emit`: the emission of a verifiable credential or tensor commitment.
- `await <event>`: parameterized by the callback event type.

Effects compose by union under the bounded join-semilattice $(P(Eff), \sqsubseteq, \sqcup, \bot)$ of §2.1. The subtyping rule is:

$$\frac{\Gamma \vdash e : A \to B \,!\, \rho' \qquad \rho' \sqsubseteq \rho}{\Gamma \vdash e : A \to B \,!\, \rho} \;\text{(T-Sub-Effect)}$$

The primitive and sequence rules feed this join structure:

$$\frac{}{\Gamma \vdash \mathtt{prim} : A \to B \,!\, \rho_{prim}} \;\text{(signature)}\;\; \text{(T-Prim)}$$

$$\frac{\Gamma \vdash a : T_a \,!\, \rho_a \qquad \Gamma, x{:}T_a \vdash b : T_b \,!\, \rho_b}{\Gamma \vdash (\mathtt{let}\ x = a\ \mathtt{in}\ b) : T_b \,!\, \rho_a \sqcup \rho_b} \;\text{(T-Seq)}$$

The declared row of a function is an upper bound on the effects its body may perform, derived by the minimum join over primitive signatures in the body and closed under T-Sub-Effect. A function whose declared effect row is a subset of another's can be substituted for the other; the reverse is not permitted. A callee that performs strictly fewer effects than declared is safe in a context expecting the larger row, because every effect the caller is prepared to handle remains handled; a callee that performs strictly more would smuggle an unhandled effect past the caller's declaration. The type checker infers the minimum effect row of an expression from its primitive calls and their composition and checks it against the declared row.

The executable checker also enforces the value boundary: `let` initializers must inhabit their annotations (with explicit linear introduction for `Linear<T>` from a `T` initializer), non-unit declared outputs require a return, program `return` expressions must inhabit the declared output type, and expression-level `match` requires a `Bool` or finite-variant scrutinee, unique known arms, full constructor coverage for known finite scrutinees, constructor-payload binders at their declared payload types, agreement of all arm result types, and a catch-all of the same result type. Thus a compiled Lex payload cannot advertise one output type while returning another, nor can it smuggle a partially typed branch under an unreachable default.

Remark (direction of effect subtyping). The subset direction matches the standard row-polymorphic discipline of Leijen's Koka (2014) and of Lindley, McBride, and McLaughlin's Frank (2017): an expression of

type `A → B ! ρ'` inhabits `A → B ! ρ` whenever `ρ' ⊆ ρ`. Lucassen and Gifford (1988) state the same principle under the vocabulary of "effect masks." The algebraic-effect line separates the effect signature from its handler; Op specializes the discipline to a fixed finite effect vocabulary and forbids open handler recovery from failed `sanctions_check`, but the subtyping direction is the literature's.

Effect-row substitution into evaluation contexts. The function-subsumption statement above has an equivalent term-level reading as an effect-row substitution rule. Let `C[·]` be an evaluation context with a single hole typed at effect row `ρ` (i.e., `C[·] : (T ! ρ) ⇒ (T' ! ρ_ctx)` for some outer row `ρ_ctx ⊇ ρ`). The substitution rule is:

```
Γ ⊢ e' : T ! ρ'                  ρ' ⊆ ρ
-------------------------------------  (Subst-Effect)
Γ ⊢ C[e'] : T' ! ρ_ctx
```

The direction is *substitute ⊆ context-expected*: the effect row of the substituted expression `e'` must be a subset of the row the context declared it would accommodate. Substitution of an `e'` whose row strictly exceeds `ρ` into a `C[·]` that promised only `ρ` is rejected, because the outer row `ρ_ctx` was computed on the assumption that the hole produced effects within `ρ`.

Concrete witness of the direction. Consider a context

```
C[·] = let x = [·] in sanctions_check(x)
```

whose hole is declared at row `ρ = {external_read}` (the hole is permitted to read one oracle) and whose outer row is `ρ_ctx = {external_read, sanctions_check}`. Substituting `e_1 = oracle_read(kyc_record)` of row `ρ_1 = {external_read}` into the hole yields `C[e_1]` at row `ρ_ctx`: the substitution is permitted because `ρ_1 = {external_read} ⊆ ρ`. Substituting `e_2 = governance_request(new_policy)` of row `ρ_2 = {governance_request}` into the same hole is rejected, because `ρ_2 ⊄ ρ`: the context was not typed to carry a governance effect past its hole, and admitting it would break Lemma 7.0.2 (Subject Reduction). In particular, a substituted `sanctions_check` effect flows from substitute to context: if the substitute introduces `sanctions_check`, the declared context must already contain `sanctions_check`; the reverse, where the context declares `sanctions_check` and the substitute does not mention it, is always admissible because the substitute's row is then strictly smaller.

Mechanized ground truth. The subset-direction discipline is the subject of the Qed-closed Coq development `op/formal/coq/EffectRow.v`, which proves that effect rows under `runion = ∪` and `subrow = ⊆` form a bounded join-semilattice (theorem `effect_row_is_bounded_join_semilattice`): `runion` is the least upper bound, the empty row is the bottom, and `subrow` is reflexive and transitive. The companion development `op/formal/coq/OpEffectMonotonicity.v` proves Lemma 7.0.3 mechanically (theorem `op_effect_monotonicity`): along any reduction trace, the accumulated effect row is subsumed by the union of the initial declared row and the active compensation bound, in the subset

direction. A paper statement that reversed the direction would fail to match either Qed-closed result; the landed paper direction matches both.

4.11 Type System Summary

Op has a fragment of a dependent type system: records, variants, lists, options, a result type, dependent function types (where the output type can mention an input value), linear resource types, and a general typestate calculus with user-defined state families, region-scoped immutable borrows, and gradual runtime refinement. The linear-resource discipline enforces that certain values (share certificates, one-time tokens, unique attestations) are consumed exactly once. The linear type constructor `Linear<T>` annotates a resource that cannot be duplicated or dropped implicitly. In the corridor fragment, `Locked<T, ω, ε>` represents a resource held in a cross-zone signed commitment, `Signed<V, ω, ε>` represents the unique local verdict witness at that index, and `Verified<ω, ε>` represents the durable pair of both sides' signed verdicts. `Locked<T, ω, ε>` has two eliminators, `commit_transfer` (consumes the lock on evidence of successful commit elsewhere) and `release_lock` (returns the underlying linear resource on evidence of abort or timeout elsewhere), and no other discharge path. The corridor history context enforces at most one active lock and at most one local signature per index `(ω, ε)`. Section 7.6 refines these forms to `T^ℓ` and `Locked<T^ℓ, ω, ε>` when information-flow labels are tracked; Section 2.6 states the general typestate calculus of which `Linear` and `Locked` are surface instances.

Base types include the kernel-native domain types: entity identifier, jurisdiction identifier, operation definition key, currency-indexed money value `Money<curr>` (integer cents at a fixed currency index `curr`), compliance domain, content digest, step identifier, callback event type, HTTP method, boolean, integer, date, timestamp, duration.

---

5. Adversarial Defenses

The preceding sections define Op's core grammar and semantics. A bytecode admitted by the kernel must also satisfy verifier-side well-formedness conditions that rule out adversarial encodings whose surface form looks compliant but whose replay evidence is underspecified. The hardening rules below are conservative: they do not add expressive power; they reject programs whose proof bundle would not let an auditor reconstruct what the operation did.

5.1 Currency-Indexed Money

Let `C` be the finite set of registered currencies, including `{USD, EUR, GBP, AED, ...}`. Op's money type is the indexed family `Money<curr>` for `curr ∈ C`; surface syntax such as `MoneyAmount` elaborates at the bytecode boundary to one such indexed type, and the index is not erasable in the typed IR. Distinct indices are distinct types:

$$curr_1 \neq curr_2 \Longrightarrow \texttt{Money<}curr_1\texttt{>} \not\equiv \texttt{Money<}curr_2\texttt{>}.$$

The only introduction form that changes the currency index is an explicit conversion primitive carrying a typed FX witness:

$$\frac{\Gamma \vdash m : \texttt{Money<USD>} \qquad \Gamma \vdash r : \texttt{FXRate(USD, EUR,}t\texttt{)}}{\Gamma \vdash \texttt{convert}_{\texttt{USD}\to\texttt{EUR}}(m, r) : \texttt{Money<EUR>}}.$$

No subsumption rule identifies `Money<USD>` with `Money<EUR>`. The proof bundle therefore records every currency swap as a first-class step with its FX witness, preventing a sanctions-evasion path in which the verifier sees only unindexed arithmetic and cannot reconstruct the currency transform.

5.2 Ensures-To-Evidence Binding

A domain verdict in source syntax is not a bare annotation. The bytecode instruction that introduces `ensures domain_d ⇝ verdict_v` must also introduce an evidence record `EvidenceEntry{domain: d, verdict: v, witness: w}` where `w : DomainEvidence<d>`. Write `WF_ens(s)` for the well-formedness predicate on a step `s`:

$$WF_{ens}(s) \iff \forall (d, v) \in Ensures(s).\, \exists w.\, \Gamma \vdash w : \texttt{DomainEvidence<}d\texttt{>} \wedge \texttt{EvidenceEntry}\{d, v, w\} \in \mu_s.$$

The verifier accepts a verdict emission only through the paired introduction rule:

$$\frac{\Gamma \vdash body : T!\rho \qquad \Gamma \vdash w : \texttt{DomainEvidence<}d\texttt{>}}{\Gamma \vdash \texttt{ensures}(d, v, w)\{body\} : T!(\rho \cup \{\texttt{proof\_emit}\})}.$$

A bytecode sequence that emits the verdict without the matching evidence value is ill-typed and rejected before execution.

5.3 PCAuth Witnesses Are Value-Bound

The PCAuth hardening already introduced at the Lex-to-Op boundary is a value-indexed attestation discipline: a discretion fill is accepted only with a witness of type `PCAuth<signer, h, v>`. The missing verifier-side statement is the binding check against the current operation input environment. Let `input_op(h)` be the value supplied to hole `h` in the operation being verified. The bytecode-verifier rule is:

$$VerifyPCAuth(op, h, v, a) \iff a : \texttt{PCAuth<signer,}h, v\texttt{>} \wedge sig\_ok(signer, a) \wedge input_{op}(h) = v.$$

The fill typing rule is admissible only when the predicate holds:

$$\frac{\Gamma \vdash v : T \qquad VerifyPCAuth(op, h, v, a)}{\Gamma \vdash \texttt{fill}(h, v, a) : T}.$$

A witness for the right signer and hole but the wrong value, or the right triple `(signer, h, v)` replayed against another operation's input vector, is rejected. This closes the bytecode attack in which a valid authority credential is replayed against a different payload.

5.4 Authenticated Oracles

An oracle consultation produces a signed object, not an untyped host read. For each oracle kind `tau`, the runtime returns `OracleResponse{data: d, signature: sig, attester: a, timestamp: t}` and the pack registers a key root `K_tau` for that oracle type. Define:

$$AuthOracle_{\tau}(r) \iff r = \texttt{OracleResponse}\{d, \mathit{sig}, a, t\} \land verify(K_{\tau}, \mathit{sig}, (d, a, t)).$$

Only authenticated responses can be projected into the program:

$$\frac{\Gamma \vdash r : \texttt{OracleResponse<}\tau\texttt{>} \qquad AuthOracle_{\tau}(r)}{\Gamma \vdash \texttt{oracle\_data}(r) : \texttt{OraclePayload<}\tau\texttt{>}}.$$

Every replay therefore checks the oracle signature against the registered public-key root for the queried oracle type; an unsigned or mis-signed host response has no Op meaning.

5.5 Substance Over Form for Compliance Primitives

Op's compliance-sensitive constructs are not name-based conventions. Let `P_comp` be the pack-versioned registry of kernel compliance primitives, with canonical lowerings `lower : P_comp → K` into kernel calls. A bytecode call `i` is a well-formed compliance invocation exactly if

$$WFCompInv(i) \iff \exists p \in P_{comp}.\, i = \texttt{invoke\_kernel}(p, \bar{v}) \land callee(i) = lower(p).$$

For named primitives such as `sanctions_check`, `corridor_translate`, and `compose_via_phi`, the verifier requires the callee digest to match the registry entry for the surface name. Equivalently,

$$\neg WFCompInv(i) \Longrightarrow \texttt{RejectBytecode}.$$

This is the substance-over-form rule: compliance semantics reside in recognized kernel calls, not in user code that imitates their names.

5.6 Reentrancy Exclusion

A compliant evaluator must refuse re-entry into an operation already in progress. The runtime maintains a lock flag indexed by operation identifier, and the type system exposes it as a typestate `OpLock<opid, s>` with `s` $\in$ `{Idle, Busy}`. Entry and exit are the only state transitions:

$$\frac{\Gamma \vdash \ell : \texttt{OpLock<opid, Idle>}}{\Gamma \vdash \texttt{enter}(opid, \ell) : \texttt{OpLock<opid, Busy>}} \qquad \frac{\Gamma \vdash \ell : \texttt{OpLock<opid, Busy>}}{\Gamma \vdash \texttt{exit}(opid, \ell) : \texttt{OpLock<opid, Idle>}}.$$

There is no typing rule whose premise contains `OpLock<opid, Busy>` and whose conclusion is a second `enter(opid,\ell)`. A second entry into the same operation identifier is therefore ill-typed, and a runtime that observes the busy flag on entry rejects the dispatch. This is the bytecode-level counterpart of the standard reentrancy guard, blocking the DAO-2016-style pattern in which control re-enters before the first invocation's state transition has closed.

5.7 Extensional-Gas Reservation for Compliance

Gas exhaustion is not allowed to starve the compliance checks whose success authorizes the later write. Let `\widehat{cost}_{san}` and `\widehat{cost}_{corr}` be the compiler's extensional upper bounds for sanctions evaluation and corridor translation on the compiled AST, including declared resumption and receipt-cardinality limits. Define the mandatory reservation

$$G_{comp}^{reserve}(P) = \widehat{cost}_{san}(P) + \widehat{cost}_{corr}(P).$$

A program is well-formed only if

$$G_{ext}^{decl}(P) \geq G_{comp}^{reserve}(P).$$

The reserved portion is committed before any write-class effect dispatches; if the declared extensional gas budget cannot cover the estimated cost of sanctions plus corridor checks, compilation fails. The effect is extensional rather than operational: the program never reaches a state in which value-moving instructions run while the compliance checks that justify them are underfunded.

6. Operational Semantics

The small-step operational semantics of Op use configurations `(e, sigma, mu, G, C)`: `e` the expression under reduction, `sigma` the environment (a map from names to values), `mu` the proof-bundle accumulator (the trace of steps committed so far), `G` the two-axis gas meter (structural + extensional), `C` the compensation stack.

6.1 Configurations and Values

A configuration is `(e, sigma, mu, G, C)`: `e` the expression under reduction, `sigma` the environment, `mu` the proof-bundle accumulator, `G` the two-axis gas meter (structural + extensional), and `C` the compensation stack, a list of pairs `(sigma_i, inverse_i)` in commit order, empty outside compensating scopes. Evaluation contexts express continuation stacking (§6.3); a serialized continuation appears only inside `Paused` tokens (§6.8) and is part of `e`, not a separate component. A value is an expression in weak head normal form: a literal, a record with all fields evaluated, a variant tag with an evaluated payload, a closure, a step-output record, or the special token `SanctionsBlocked`.

Frame-rule convention. Throughout §6, reduction rules mention only the configuration components they mutate; elided components are threaded from premise to conclusion unchanged. Rules that touch `mu`, `C`, or `G` always make that touch explicit. A rule written without `C` preserves `C`; a rule written without `G` preserves the structural-gas balance modulo the one-unit-per-rule charge of §6.2; a rule written without `mu` leaves `mu` unextended. Subject reduction (§7.1) is stated over the full quintuple and relies on this convention to interpret the rules as quintuple-indexed.

6.2 Structural Gas

The gas meter `G` is a pair `(G.structural, G.extensional) : Nat × Nat` on disjoint fields. Every small-step reduction consumes exactly one unit of structural gas, on `G.structural` only:

```
G |- e, sigma, mu, C  --tick-->  e', sigma', mu', C'     if G.structural ⩾ 1
(the step produces G' with G'.structural = G.structural - 1;
 the G.extensional field is set by the individual rule)
```

The one-unit structural charge is invariant across §§6.4-6.12 and §6.15. Rules that write `G - 1 - gas_ext(s)` in their conclusion decrement the structural field by one and additionally decrement the extensional field by `gas_ext(s)`; they do not charge `1 + gas_ext(s)` to the structural field. Rules that write `G - 1` alone leave the extensional field unchanged. The program's initial structural-gas balance is the compiler-computed bound `B` from §4.9: the count of reduction-rule-applying forms in the compiled AST multiplied by the declared maximum resumption multiplicities across `await` sites.

Reduction halts with `OutOfStructuralGas` on budget exhaustion. The halt is observable in the proof bundle, not a silent failure.

6.3 Evaluation Contexts

An evaluation context `E` is an expression with a single hole `[·]`, generated by:

```
E ::= [·]
    | let x : T = E in e
    | E ; t
    | par { a = E; b = e_b } | par { a = v_a; b = E }
    | choose { when E -> b; ... } | choose { when v_1 -> b_1; ...; when E -> b_k; ... }
    | step s { E } compensate { inverse }
    | in j { E }
```

Reduction is closed under evaluation contexts: if `(e, sigma, mu, G, C) --> (e', sigma', mu', G', C')` by a base rule, then `(E[e], sigma, mu, G, C) --> (E[e'], sigma', mu', G', C')`.

6.4 Let Binding

```
                v is a value
-------------------------------------------- (E-Let)
(let x : T = v in e, sigma, mu, G)
  --> (e, sigma[x |-> v], mu, G - 1)
```

6.5 Sequential Step Composition

```
            s is a committed step with value v, trace-entry c = commit(s, v)
------------------------------------------------------------------------ (E-Seq)
(s ; t, sigma, mu, G)
  --> (t, sigma[s.name |-> v], mu ++ [c], G - 1 - gas_ext(s))
```

The `commit(s, v)` entry in `mu` records the step's identifier, its input-value hash, its output-value hash, its effect row, and its service-call receipt if one was produced.

6.6 Parallel Composition

```
par { a = e_a; b = e_b }
--------------------------------- (Par-1)
(e_a, sigma, mu, G, C) and (e_b, sigma, mu, G, C) independently reduce,
and their final values are joined into a record { a = v_a; b = v_b }
```

Par reduces branches in source order: the left branch reduces to a value first, then the right branch, with a shared pre-state and disjoint bundle-extensions. The reduction rule:

```
    (e_a, sigma, mu, G, []) -->* (v_a, sigma, mu ++ mu_a, G_a, C_a)
```

```
   (e_b, sigma, mu ++ mu_a, G_a, []) -->* (v_b, sigma, mu ++ mu_a ++ mu_b, G_b, C_b)
------------------------------------------------------------------------ (E-Par)
(par { a = e_a; b = e_b }, sigma, mu, G, C)
  --> ({ a = v_a; b = v_b }, sigma, mu ++ mu_a ++ mu_b, G_b - 1,
        C ++ C_a ++ C_b)
```

Each branch runs against an empty compensation stack; on join, the per-branch stacks concatenate onto the enclosing `C` in source order. An abort within branch `a` drains `C_a` alone before propagating; a post-join abort drains `C ++ C_a ++ C_b` in commit-reverse order. Source-order commit makes `mu` a deterministic function of the program and inputs. Implementations may reduce branches concurrently when the confluence lemma (§7.1a) applies; the lemma discharges what would otherwise be an implementation obligation.

Open lemma schema (Par-Confluence). Let `par { a = e_a; b = e_b }` be a well-typed parallel composition: the type checker has rejected cross-branch data dependence and the branches' extensional-gas charges sum to at most the enclosing program's extensional-gas budget. Let `c_serial` be the post-state of source-order serial execution and `c_concurrent` the post-state of any interleaving whose reductions preserve branch-internal order. The target statement is: (i) `sigma_serial = sigma_concurrent` pointwise; (ii) `mu_concurrent` is a branch-internal-order-preserving permutation of `mu_serial`; (iii) `G_serial = G_concurrent` on both axes; (iv) `C_serial` and `C_concurrent` agree up to the same permutation as (ii). After the source-order canonicalization pass on `mu` at join time, `mu_concurrent = mu_serial` bit-for-bit.

Proof sketch target. Structural induction on the number of branch-interleaving swaps between the concurrent trace and the serial trace. Each swap either (a) exchanges two independent reductions, which commute under the disjoint-data-dependence condition and the shared-budget premise, or (b) commutes a branch-internal reduction past a branch-join, which leaves branch-internal commit order unchanged. Summing swaps reduces the concurrent trace to the serial trace. The canonicalization pass at join time (source-order commit of `mu_a ++ mu_b`) collapses the permutation into byte-identical order. This remains a proof obligation for the full semantics.

6.7 Guarded Choice

```
choose { when g_1 -> b_1; when g_2 -> b_2; else -> b_e }
------------------------------------------------------- (Choose-1)
evaluate g_1 in sigma:
  if true, reduce b_1
  if false, evaluate g_2:
    if true, reduce b_2
    else reduce b_e
```

Guards evaluate in source order. The first guard to return `true` selects its block. The `else` branch is mandatory when the preceding guards do not cover the input space; the compiler checks coverage against a finitary guard grammar.

6.8 Suspension

```
await E within d
---------------------------------  (Await-1)
serialize the surrounding evaluation context into a continuation blob K_blob;
emit a Paused(E, K_blob, event_budget = d) token, where `d` is the callback-event-count offs
halt.
```

The suspension rule produces the `Paused` token. The runtime carries it as an external object: persisted in the proof bundle, published to the callback registry, matched against incoming events. When a matching event arrives with payload `p : T` (where `T` is the declared payload type of `E`), reduction resumes:

```
(Paused(E, K_blob, _), event E with payload p)
-----------------------------------------------  (Resume-1)
deserialize K_blob into an evaluation context E' with hole;
continue reduction from (E'[p], sigma, mu, G', C)
```

If the event budget is exhausted before a matching event arrives, the reduction resumes with a timeout, which triggers the declared failure policy (Section 6.10).

6.9 Compensation

The compensation stack `C` (§6.1) is empty outside compensating scopes and accumulates `(sigma_i, inverse_i)` pairs in commit order as compensating steps succeed. The compensation construct reduces by:

```
           (body, sigma, mu, G, C) -->* (v, sigma', mu', G', C)
------------------------------------------------------------------- (E-Comp-Success)
(step s { body } compensate { inverse }, sigma, mu, G, C)
  --> (v, sigma', mu' ++ [commit(s, v)], G' - 1,
       C ++ [(sigma', inverse)])

           (body, sigma, mu, G, C) -->* Failure(err, mu_f, G_f)
------------------------------------------------------------------- (E-Comp-Fail)
(step s { body } compensate { inverse }, sigma, mu, G, C)
  --> Abort(err, mu_f, G_f, C)
```

Abort triggers the compensation pass:

```
               C = C' ++ [(sigma_n, inverse_n)]
```

```
    (inverse_n, sigma_n, mu, G, []) -->* (_, _, mu_n, G_n, _)
 ------------------------------------------------------------------- (E-Abort-Pop)
 Abort(err, mu, G, C)
   --> Abort(err, mu ++ [inverse_commit(inverse_n, mu_n)], G_n, C')

 ------------------------------------------------- (E-Abort-Done)
Abort(err, mu, G, [])  -->  Halted(err, mu)
```

Idempotency obligation. An `inverse` block must satisfy, for every post-state `sigma'` of its forward body, `inverse(inverse(sigma')) = inverse(sigma')` under the external-service semantics. The type checker rejects any `inverse` whose effect row does not include the forward step's effect row; it does not check extensional idempotency, a primitive-library obligation discharged against the program-logic layer of §11 and completed, for primitive libraries, by the work left open in §14.9.

6.10 Failure Policies

A step carries an `on_failure` policy, one of: `cancel_operation` (abort the whole program, run full compensation), `rollback` (run compensation for this step and continue), `skip` (mark the step skipped, continue), `retry` (re-attempt under a retry schedule), or `continue` (record the failure, proceed as if successful; permitted only for steps whose declared effect is compensable at no cost). The compiler rejects `continue` for any step whose effect row contains `sovereign_write`, `identity_mutation`, or `fiscal_transfer`; a continue-on-failure sovereign write would silently abandon a compliance violation, which the effect-system law forbids.

6.11 Corridor-Indexed Typestates

For the corridor fragment we refine the ordinary typing judgment to $\Gamma \mid \kappa \vdash e : T \,!\, \rho$, where κ is a finite history map keyed by local side `X` and corridor index (ω, ε). Each cell records three static facts: whether side `X` presently owns an active lock at (ω, ε), whether `X` has already signed a verdict at that index, and the event-count deadline attached to that lock. The predicate `durable_X(`κ`, `ω`, `ε`, {s_A, s_B})` introduced below is a predicate on κ alone; §7.0's κ-μ coherence obligation ensures the static record and the dynamic proof bundle do not diverge. When κ is irrelevant we elide it and write the shorter judgment $\Gamma \vdash e : T \,!\, \rho$ used elsewhere.

Morphism constructors. Read each rule below as a constructor in the indexed typestate category of §2.4: the premise's typestate and history cells are the object under which the morphism begins; the conclusion's typestate is the object at which it lands; and the side-conditions on κ restrict the constructor's domain.

```
Γ | κ ⊢ v : Linear<T> ! ∅      active_lock_X(κ, ω, ε) = ∅
 ------------------------------------------------------  (T-Lock)
Γ | κ ⊢ lock(v, ω, ε, d) : Locked<T, ω, ε> ! {proof_emit}
```

Successful typing of `lock` extends κ with the singleton entry `active_lock_X(`ω`, `ε`) = d`. Because no rule duplicates, weakens, or rewrites that cell, a second active lock at the same side and index is untypeable.

Proposition 6.11.1 (Lock uniqueness, I1). For every well-typed corridor program and every reachable typing context κ: at each side `X` and each index `(`ω`, `ε`)`, at most one active-lock cell is present in κ. Proof: structural induction on the typing derivation; the only cell-introducing rule is T-Lock, whose premise `active_lock_X(`κ`, `ω`, `ε`) = `$\emptyset$ rules out a prior cell at the same `(X, `ω`, `ε`)`; no other rule adds lock cells; removal rules (E-Lock-Timeout, E-Commit, E-Release) can only delete. Mechanized as `invariant_I1` in `github.com/momentum-sez/op/formal/coq/BSCInvariants.v` (theorem `op_lock_preserves_I1` and its companions across the full operation family).

```
Γ | κ ⊢ v : V ! ∅      history_of_X(κ, ω, ε) = ∅
------------------------------------------------  (T-Sign)
Γ | κ ⊢ sign(v, ω, ε) : Signed<V, ω, ε> ! {proof_emit}
```

The successful derivation records the signed digest in `history_of_X`. Any second signature by the same side at the same `(`ω`, `ε`)` fails the emptiness premise and is therefore a typing error.

Proposition 6.11.2 (Verdict uniqueness, I2). For every well-typed corridor program and every reachable typing context κ: at each side `X` and each index `(`ω`, `ε`)`, at most one signed-verdict cell is present in κ. The history context carries uniqueness. Two distinct values of type `Signed<V, `ω`, `ε`>` with the same `(side, `ω`, `ε`)` may be syntactically formable; no well-typed derivation produces both in a single κ. Proof: same structural-induction pattern as 6.11.1, with T-Sign as the sole cell-introducing rule and no removal rule for signature cells. Mechanized as `invariant_I2` in `BSCInvariants.v`.

```
Γ | κ ⊢ s_A : Signed<V_A, ω, ε> ! ∅
Γ | κ ⊢ s_B : Signed<V_B, ω, ε> ! ∅
durable_X(κ, ω, ε, {s_A, s_B})
------------------------------------------------------------  (T-Verify)
Γ | κ ⊢ record(s_A, s_B, ω, ε) : Verified<ω, ε> ! {proof_emit}
```

The predicate `durable_X(`κ`, `ω`, `ε`, {s_A, s_B})` is defined on κ as the conjunction `has_sign_X(SideA, (`ω`, `ε`), `κ`) ∧ has_sign_X(SideB, (`ω`, `ε`), `κ`)`: both signed-verdict cells are already present at side `X`'s local history. Under §7.0's κ-μ coherence, presence in κ is equivalent to presence in the local proof bundle μ; entry to `Verified<`ω`, `ε`>` is therefore durable by construction on either reading.

Proposition 6.11.3 (Verified durability, I3). If `Γ | `κ` ⊢ record(s_A, s_B, `ω`, `ε`) : Verified<`ω`, `ε`> ! `ρ, then κ already contains signed-verdict cells at index `(`ω`, `ε`)` for both `SideA` and `SideB`. Proof: inversion on T-Verify yields the `durable_X` premise, which is the conjunction above. Mechanized as `invariant_I3` in `BSCInvariants.v` (theorem `op_verify_preserves_I3`, using precondition `op_verify_pre`).

```
Γ | κ ⊢ s_1 : Signed<V, ω, ε> ! ∅
Γ | κ ⊢ s_2 : Signed<V, ω, ε> ! ∅
signer(s_1) = signer(s_2) = Z      payload(s_1) ≠ payload(s_2)
-------------------------------------------------------------  (T-Blame)
Γ | κ ⊢ blame(s_1, s_2) : Blame<Z, ω, ε> ! {proof_emit}
```

Remote equivocation is not reified as a second verified state; once the conflicting artifacts are imported and satisfy the same-signer/same-index conflict predicate, it is reified as a typed blame witness naming the signer that produced the conflicting pair. Categorically, T-Verify and T-Blame cover disjoint preconditions for the two cases they name: T-Verify applies when both sides' payloads are consistent with a well-formed joint verdict and the local durability predicate holds, while T-Blame applies when a same-signer pair disagrees. The Qed-closed kappa core proves that this blame precondition violates verdict uniqueness (`op_blame_pre_violates_I2` in `BSCInvariants.v`); it does not yet prove watcher extraction from remote evidence. Missing finality evidence, absent peer signatures, malformed witness encodings, and non-same-signer disagreement are typed obstructions rather than members of either constructor. The totality theorem for attempted joint-verdict formation is therefore open.

```
Γ | κ ⊢ l : Locked<T, ω, ε> ! ∅      deadline_X(κ, ω, ε) = d
-------------------------------------------------------------  (T-Lock-Timeout)
Γ | κ ⊢ timeout_check(l, ω, ε) : Result<Linear<T>, Locked<T, ω, ε>> ! {proof_emit}

events_since_lock_X(mu, ω, ε) ⩾ deadline_X(κ, ω, ε)
-------------------------------------------------------------------------  (E-Lock-Timeo
(timeout_check(l, ω, ε), sigma, mu, G, C)
  --> (Ok(release_lock(l, timeout_witness(ω, ε))),
       sigma,
       mu ++ [lock_timeout(ω, ε)],
       G - 1,
       C)
```

When the bound is not yet met, `timeout_check` reduces to `Err(l)` and leaves κ unchanged. The deadline counts callback, receipt, or locally generated watchdog events after lock introduction, not wall-clock time. Partial synchrony in the sense of Dwork, Lynch, and Stockmeyer (1988) is therefore not enough by itself: the host must also deliver a local tick or failure-detector event that advances the event count. Conditional on that event source, either the remote signed witness arrives before the bound or the lock is released by this rule.

Release as index-forgetting projection. `release_lock` and E-Lock-Timeout both consume a `Locked<T, ω, ε>` and yield a `Linear<T>` whose carrier inhabitant is the unchanged value. In the indexed-category presentation of §2.4, each is a morphism out of the fiber at `(ω, ε)` into the fiber of `Linear` that forgets the index but preserves the carrier. `release_lock` is invoked when a counterparty

abort witness arrives; E-Lock-Timeout is invoked when the event-count deadline expires. The two paths have distinct proof-bundle extensions (`lock_released(ω, ε)` and `lock_timeout(ω, ε)` respectively) but land at the same categorical destination. The mechanized operation `op_timeout` and its alias `op_discharge` in `BSCInvariants.v` model both: what is invariant across them is the removal of the `(side, ω, ε)` lock cell from κ, which is what Propositions 6.11.1-6.11.3 require be preserved.

6.12 Sanctions Dominance

Sanctions dominance has a static and a dynamic half. The static half is SD (§2.1): every write-class step must be graph-dominated by a `sanctions_check` on every source-to-step path in the program DAG. The dynamic half is the fail-closed transition below, which fires when a checked subject resolves to an applicable non-compliant verdict at runtime.

```
   sanctions_check in rho_e          host_sanctions(s) = NonCompliant
 ------------------------------------------------------------------------  (Sanctions-Hard-Bl
   (e, sigma, mu, G, C) ---> Abort(SanctionsBlocked, mu ++ [sanctions_block(s, rho_e)], G, C)
```

Here `rho_e` is the declared effect row of `e`, inherited from program text and already checked by SD. `host_sanctions` is the deterministic host oracle from the subject `s` of the pending write-class step to the sanctions verdict, content-addressed and logged before the rule fires. The subject is determined by the typed step input: for `fiscal_transfer(from, to, amount)` the subjects are `{from, to}`; for `sovereign_write` on entity `e` the subject is `e`; for `identity_mutation` on principal `p` the subject is `p`.

The block is a reduction, not a typing rule, because the verdict is a runtime oracle artifact. The typing system proves that a reachable write-class redex must already carry `sanctions_check`; the reduction consumes the oracle result and transitions to an absorbing failure when the applicable result is non-compliant.

The rule fires an `Abort` whose error token is `SanctionsBlocked`, whose proof-bundle extension records `(s, rho_e)`, and whose compensation stack `C` is drained by the E-Comp-Fail pass. `Abort` is not itself terminal: E-Comp-Fail pops each `(sigma_n, inverse_n)` and reduces it against `sigma_n`, appending the resulting trace to `mu`. A compensation inverse whose own sanctions check would produce `NonCompliant` emits the terminal `InversionSanctionsBlocked`, halting the drain and leaving residual forward commits marked for operator reconciliation. The drain's final configuration is `SanctionsBlocked(mu_final)`, where `mu_final` includes the sanctions-block entry, every committed forward entry prior to the block, and every reversible compensation entry. `SanctionsBlocked(mu)` is terminal; no reduction rule produces a downstream value from it. The design privileges sanctions enforcement over automatic reversal: a sanctions block that prevents reversal is a regulatory incident, not an automatic retry.

One structural exception: `create.entity` (creating a new entity that does not yet exist) may begin before sanctions screening. Screening happens post-flight; the `SanctionsBlocked` outcome, if it arises after applicability evidence is evaluated, blocks the creation's completion and inverts the partial creation through

the pre-registered compensation. In §6.6, `SanctionsBlocked` is typed at the bottom element for every security label, so the fail-closed result propagates through labeled continuations at all labels rather than only the blocked subject's original label.

6.13 Effect Propagation

Effect propagation under composition is a derived consequence of the typing rules. For sequential composition:

```
  Γ ⊢ a : T_a ! ρ_a        Γ, x:T_a ⊢ b : T_b ! ρ_b
-------------------------------------------------------  (T-Seq)
  Γ ⊢ (let x = a in b) : T_b ! ρ_a ⊔ ρ_b
```

The conclusion row $\rho\_a \sqcup \rho\_b$ is forced by the join's upper-bound property: $\rho\_a \sqsubseteq \rho\_a \sqcup \rho\_b$ and $\rho\_b \sqsubseteq \rho\_a \sqcup \rho\_b$. Parallel composition and guarded choice join branch rows analogously (§6.6, §6.7). Compensation joins the inverse's declared row into the forward row as a lower bound on the compensated step (§2.2, §3.3). Operationally, under any reduction trace, the accumulated row is subsumed by the union of the initial declared row and the active compensation bound. The Qed-closed companion theorem is `op_effect_monotonicity`; on termination, the final accumulated row is recorded in the proof bundle for post-flight compliance evaluation.

6.14 Termination

Reduction phases. A phase is a maximal reduction sequence under the small-step relation `-->` that ends in a phase-terminal configuration: a value, `Paused(E, K_blob, b)`, `Halted(err, mu)`, `SanctionsBlocked(mu)`, `AuthorizationBlocked(mu)`, `OutOfStructuralGas(mu)`, `OutOfCompensationGas(mu)`, or `Timeout(mu)`. Resumption from `Paused` enters a new phase whose initial structural-gas balance is the post-suspension balance carried forward from the prior phase.

Target theorem (single-phase termination). Let `P` be a well-typed Op program whose initial configuration has finite structural-gas balance `B`. Every `-->` reduction sequence starting from that configuration is finite and terminates in a phase-terminal configuration.

Proof sketch. Define `mu_struct(c) = c.G.structural`. Every base reduction rule of §§6.4-6.12 and every terminal-trap rule of §6.15 strictly decreases `mu_struct` by at least one under the frame-rule convention of §6.1 and the gas rule of §6.2. Evaluation contexts preserve the base-rule decrement. Since `<` on `Nat` is well-founded, no infinite strictly descending reduction chain exists; any phase has length at most its initial structural-gas balance. Exhausting the balance before a grammar-level terminal forces a gas terminal; otherwise the phase ends in a value, suspension, halt, sanctions block, authorization block, or timeout.

The total reduction count across resumption phases is bounded by the compiler's structural-gas budget `B`, because resumption carries forward the remaining balance and every phase consumes at least one structural-

gas unit before making progress. Bounded iteration is not a primitive; it is available only as finite expansion of participant or compliance-domain lists at compile time.

Mechanized ground truth. The parametric single-phase theorem is Qed-closed in `op/formal/coq/GasTermination.v` for any semantics satisfying the interface axiom `gas_decreases : forall c c', step c c' → gas c' < gas c`. The concrete toy instance in `OpConcreteAST.v` discharges that axiom for a nine-constructor gasified AST. Instantiating the parametric theorem for the full Op operational semantics remains an open §14.1 obligation: the paper-level proof reduces it to the rule-by-rule fact that every reduction rule decrements `G.structural`.

6.15 Terminal Failure Rules

Gas exhaustion, event-budget exhaustion, and insufficient compensation reserve each produce terminal configurations:

```
          G.structural = 0     e is not already a value or phase terminal
-------------------------------------------------------------- (E-OOG)
(e, sigma, mu, G, C)
  --> OutOfStructuralGas(mu)
```

E-OOG fires when the structural balance reaches zero on a non-terminal configuration. Zero-balance configurations already holding a value or phase terminal do not reduce.

```
          b = 0  (event budget exhausted in a Paused continuation)
-------------------------------------------------------------- (E-Timeout)
(Paused(E, K_blob, b), sigma, mu, G, C)
  --> Timeout(mu)
```

E-Timeout fires when the event-budget counter on a paused continuation reaches zero without a matching resumption event. The deadline counts callback or receipt events, not wall-clock time (§6.1).

```
               remaining G insufficient for compensation_budget
------------------------------------------------------------------- (E-OOCG)
Abort(err, mu, G, C) with exhausted compensation budget
  --> OutOfCompensationGas(mu)
```

E-OOCG is the compensation-phase analogue of E-OOG. During an abort drain, each popped inverse reduces as a nested phase charging the shared structural-gas balance of `G_total`; if cumulative compensation charge exceeds the declared `compensation_budget`, the drain terminates in `OutOfCompensationGas(mu)` and residual forward commits remain marked for operator reconciliation.

Each rule produces a terminal configuration that types at `⊥_T` per the bottom-element rule of §7.0. Downstream `bind` or `match` propagation therefore preserves subject reduction across terminal outcomes.

7. Soundness and Conservation

Op's conservation invariants depend on type-soundness lemmas stated below, with proof sketches in the structural-induction style standard for small-step semantics.

7.0 Type Soundness

Typing judgment. The load-bearing judgment threads a corridor history from pre- to post-state:

```
Γ | κ ⊢ e : T ! ρ ⇒ κ'
```

read as "under variable-and-credential context Γ and input corridor history κ, expression `e` has type `T`, emits effect row ρ, and yields output corridor history κ`'`." The components:

1. Γ is a finite partial function from names to pairs `(T, ω)` where `T` is a type and ω a (possibly empty) credential set, disjoint in variables and principals.
2. $\kappa \in$ `K` is the finite history map of §6.11, keyed by local side `X` and corridor index `(ω, ε)`, whose cells record active-lock ownership, signed-verdict presence with payload digest, and lock deadlines.
3. $\rho \subseteq$ `Σ_eff` is a finite subset of the closed effect vocabulary `Σ_eff` of §4.10, ordered by inclusion; inclusion is a bounded join-semilattice with $\emptyset$ least and union as join , the finite-vocabulary specialization of the row calculus of Lucassen and Gifford (1988).
4. κ`'` is the post-expression history, determined by the structural rule applied to `e`.

When a rule does not touch κ we elide it and write the shorter judgment `Γ ⊢ e : T !` ρ; this is definitionally equivalent to `Γ |` κ `⊢ e : T !` $\rho \Rightarrow \kappa$ for every κ. Linearity follows Wadler (1990); the split-context lemma of §4.10 is recovered by restricting Γ to its linear subcontext and requiring that subcontext to split multiplicatively across binary forms. The system is bidirectional in the Pierce-Turner (2000) sense: every elimination form is checked against a type inferred from its principal premise; every introduction form is checked against an expected type. Checking mode and synthesis mode agree on typability. Section 7.6 refines the base judgment to `Γ; pc; A; Ξ ⊢ e : T^`ℓ `!` $\rho \Rightarrow$ `Ξ'` when security labels, a program-counter label, explicit authority, and a labeled compliance context are tracked; there κ is the restriction of Ξ to its non-label component.

Structure of the compliance-context space `K`. A point $\kappa \in$ `K` is a finite partial function from triples `(X, ω, ε)` $\in$ `Sides × Corridors × Ends` to cells `(`ℓ`, s, d)` $\in$ `{`$\perp$`, active} × ({`$\perp$`} ∪ {signed(h) : h ∈ Digests}) × (`$\mathbb{N} \cup \{\infty\}$`)`, where ℓ records active-lock presence, `s` records whether side `X` has signed a verdict at that index (and with which payload digest `h`), and `d` is the event-count deadline (∞ denoting "no deadline set"). Read $\sqsubseteq$ as the information-refinement order: κ`_1` $\sqsubseteq$ κ`_2` means "κ`_1` records at least everything κ`_2` records, and possibly more." Define $\sqsubseteq$ cellwise: on the lock field, `active` $\sqsubseteq \perp$ (an active lock is a strict refinement of "no recorded lock"); on the signature field,

signed(h) $\sqsubseteq$ $\bot$ for every h, and distinct digests signed(h) $\neq$ signed(h') are incomparable; on the deadline field, d_1 $\sqsubseteq$ d_2 $\Leftrightarrow$ d_1 $\leq$ d_2 (a tighter finite bound refines a looser one, and every d $\in$ $\mathbb{N}$ refines ∞); each field compares reflexively to itself. Extend $\sqsubseteq$ pointwise across keys, treating undefined cells as (⊥, ⊥, ∞). Under $\sqsubseteq$, K is a partial order with greatest element $\top$_K, the everywhere-undefined map, equivalently the constant (⊥, ⊥, ∞) map, and no universal least element (incomparable signed digests at the same key witness incomparability rather than a common bottom). When both sides agree on every signed digest the pointwise infimum defines a partial meet κ_1 $\sqcap$ κ_2; when they disagree, the derivation fails at T-Blame of §6.11 rather than producing a meet. K is therefore a bounded partial meet-semilattice with top. Every structural rule of §6.11 that touches κ does so by cell-insertion, promoting $\bot$ to active (T-Lock), $\bot$ to signed(h) (T-Sign), or recording a Verified pair (T-Verify), and each insertion takes κ strictly downward in $\sqsubseteq$ toward a more informative history. No structural rule weakens a set cell and no rule erases a cell.

κ-threading functoriality. Fix a derivation $\mathcal{D}$ of Γ | κ $\vdash$ e : T ! ρ $\Rightarrow$ κ'. The cell-insertion discipline gives κ' $\sqsubseteq$ κ whenever $\mathcal{D}$ exists. The assignment κ $\mapsto$ κ' induced by $\mathcal{D}$ is a monotone partial function $\mathcal{T}$_{$\mathcal{D}$} : K $\rightharpoonup$ K, defined on the downward-closed sub-poset $\downarrow\kappa$ (histories at least as informative as the derivation's declared input) and taking values in $\downarrow\kappa$', itself a sub-poset of $\downarrow\kappa$. For sequential composition e_1; e_2 with derivations $\mathcal{D}$_1 on e_1 and $\mathcal{D}$_2 on e_2 whose induced maps are $\mathcal{T}$_{$\mathcal{D}$_1} and $\mathcal{T}$_{$\mathcal{D}$_2}, the induced map of the composite derivation is $\mathcal{T}$_{$\mathcal{D}$_2} $\circ$ $\mathcal{T}$_{$\mathcal{D}$_1}: ordinary function composition, associative, with the identity expression inducing id_K. This is the precise content of the "κ-threading rule defines a functor" claim: the Op derivation system, restricted to sequential composition, presents a forgetful functor from well-typed Op programs to monotone maps on (K, $\sqsubseteq$), and every basic structural rule of §6.11 is a generating arrow that is strictly monotone on the single cell it inserts. The two-point quotient identifying every post-sanctions cell with an absorbing KBlocked and every otherwise-typed cell with KClean is the core-fragment compliance lattice of OpMetaTheory.v; core_compliance_context_monotonicity proves monotone descent on that quotient for the mechanized core reduction.

Proof obligation 7.0.1 (Progress). Let τ $\in$ {Paused(E, K_blob, b), Halted(err, mu), SanctionsBlocked(mu), AuthorizationBlocked(mu), OutOfStructuralGas(mu), OutOfCompensationGas(mu), Timeout(mu)} denote the terminal-configuration set. Suppose $\emptyset$ | κ $\vdash$ e : T ! ρ $\Rightarrow$ κ' and e is neither a value nor a terminal in τ. The target full-judgment statement is that for every well-formed initial configuration (e, σ, μ, G, C) there exist (e', σ', μ', G', C') with (e, σ, μ, G, C) $\rightarrow$ (e', σ, μ', G', C'). Well-formedness of the initial configuration requires σ to agree with Γ on every bound name, μ to be a prefix-ordered append-only sequence, and C to be compensation-well-formed in the sense of Lemma 7.0.2. The induction proceeds over the typing derivation of e with case analysis on its principal form; the canonical-forms lemma at each type extracts a structural head pattern that either matches a reduction-rule redex or exposes a subexpression that steps by the induction hypothesis. The core-fragment case is mechanized as op_progress in

`OpProgressSubject.v` (`github.com/momentum-sez/op`); extension to the full judgment with κ-threading is listed under §14.1.

Proof obligation 7.0.2 (Subject Reduction). Suppose

1. Γ `|` $\kappa \vdash$ `e : T !` $\rho \Rightarrow \kappa$`'`;
2. The compensation stack `C` is well-formed: every pair `(`σ`_i, inverse_i)` $\in$ `C` satisfies Γ`_i |` κ`_i` $\vdash$ `inverse_i : Unit !` ρ`_i` $\Rightarrow$ κ`_i''` with ρ`_i` $\supseteq$ ρ`_{forward_i}` per the §6.9 compensation typing obligation, and κ`_i` is the history recorded at the moment the forward step `i` committed its pair into `C` (so κ`_i` $\sqsubseteq$ κ`_{forward,pre}` in the refinement order);
3. `(e,` σ`,` μ`, G, C)` $\rightarrow$ `(e',` σ`',` μ`', G', C')`.

Then there exist Γ`'` $\supseteq$ Γ and κ`_step` $\sqsubseteq$ κ such that Γ`' |` κ`_step` $\vdash$ `e' : T !` ρ`''` $\Rightarrow$ κ`''`, where ρ`''` $\subseteq$ ρ, κ`''` `=` κ`'`, and `C'` is compensation-well-formed. The context extension Γ`'` $\supseteq$ Γ records names bound by E-Let and E-Seq, the only reduction rules extending σ. The refinement κ`_step` $\sqsubseteq$ κ follows from the cell-insertion discipline of §6.11: every rule fired either leaves κ unchanged (non-corridor reductions, so κ`_step =` κ) or refines exactly one cell toward a more informative value (a corridor rule, so κ`_step` $\sqsubset$ κ with a single cell distinct). Equality κ`''` `=` κ`'` of the output history reflects that the total collection of cells set by the full derivation of `e` coincides with the cells set by the step just fired (giving κ`_step`) plus the cells remaining to be set inside `e'` (which take κ`_step` to κ`''`): these together equal κ`'`. The core-fragment case, `has_type G e T` preserved across `step e e'`, is mechanized as `op_subject_reduction` in `OpProgressSubject.v`. Pointwise descent of κ on the two-point lattice `{KBlocked, KClean}` is mechanized as `core_compliance_context_monotonicity` in `OpMetaTheory.v`. The refined judgment with full `K` is listed under §14.1.

Terminal failure tokens are typed at the bottom element of the type lattice: for every τ $\in$ `{Halted(err, mu), SanctionsBlocked(mu), AuthorizationBlocked(mu), OutOfStructuralGas(mu), OutOfCompensationGas(mu), Timeout(mu), Abort(err, mu, G, C)}` and every type `T`, the judgment Γ `|` $\kappa \vdash \tau$ `: ⊥_T !` $\rho \Rightarrow \kappa$ holds, where `⊥_T` is the least element of the type lattice under its elimination-preorder (no eliminator at `T` produces a non-`⊥_T` value from `⊥_T`). Subject Reduction then preserves typing across E-rules that produce terminal tokens: a typed pre-configuration reducing to a terminal token still types at `⊥_T !` ρ`'` $\Rightarrow \kappa$ with ρ`'` $\subseteq \rho$, and the history coordinate is preserved unchanged on the terminal step (the cell-insertion of §6.11 that would fire is instead absorbed by the terminal rule, per §6.12).

Lemma 7.0.3 (Effect Monotonicity). The effect row accumulated along a reduction trace ρ`_trace` satisfies $\rho_{\text{trace}} \subseteq \rho_{\text{declared}} \cup \bigcup_i \rho_{\text{inverse}_i}$, where ρ`_declared` is the statically declared row of the program and each ρ`_{inverse_i}` is the effect row of a compensation inverse executed on the trace. In the absence of compensation, ρ`_trace` $\subseteq$ ρ`_declared` exactly. The corresponding core result is mechanized; the full threaded judgment remains part of the §14.1 proof plan.

Proof sketch (Lemmas 7.0.1-7.0.3). Structural induction on the typing derivation, case analysis on the reduction rule applied. The linear fragment requires the split-environment lemma of §4.10 (every linear variable is used exactly once in a well-typed expression). The effect-row case requires that every reduction rule's ρ' equal ρ (non-effectful reductions, such as E-Let and E-Seq) or be a strict subset (an effectful primitive dispatches and discharges one effect of the declared row). §14.1 targets the mechanization of the full threaded judgment.

Open theorem schema 7.0.4 (Compliance-carrying soundness). Let P be an Op program such that $\emptyset$ | κ_0 ⊢ P : T ! ρ under the compliance-carrying typing judgment, with κ_0 the top of the compliance-context lattice K. The target statement is that every reduction trace (P, σ_0, μ_0, G_0, C_0) →* (v, σ', μ', G', C') satisfies the following conjoint invariant:

1. *(Type preservation in context.)* There exist κ' with $\emptyset$ | κ' ⊢ v : T ! ρ' and ρ' ⊆ ρ.
2. *(Monotone compliance accumulation.)* κ' ⊑ κ_0 in the lattice order of K (the compliance context descends under meet with every discharged check and never rises spontaneously).
3. *(Sanctions absorption.)* If at any intermediate step κ_i ⊑ ⊥_sanc, then the trace ends in a SanctionsBlocked terminal with the proof bundle μ recording the triggering sanctions_check entry; no downstream continuation restores κ_i above ⊥_sanc.
4. *(Resource linearity and ownership.)* Every value of type Linear<T> or Locked<T, ω, ε> introduced on the trace is consumed exactly once, and every state mutation committed to μ names the authorized writer required by its primitive.
5. *(Audit completeness.)* The proof bundle μ' extends μ_0 content-addressably and contains, for every discharged compliance check check(φ_i) fired on the trace, an entry witnessing the value-bound PCAuth attestation, the jurisdiction, the decision time, and the resulting compliance-context refinement.

Proof sketch. Progress, subject reduction, and effect monotonicity give (1) by induction on the reduction rule: the compliance-context coordinate κ updates pointwise under the stepwise rule for each primitive dispatch, and the rule for check(φ) refines κ by ⊓ ⌈φ⌉ (monotone by construction). (2) follows because every update is a meet in K, hence below its predecessor. (3) is the sanctions-dominance law: the semantics dispatches every primitive whose effect row contains sanctions_check to SanctionsBlocked whenever the returned verdict is Blocked, and the terminal is bottom-propagating at every type. (4) is the existing resource-linearity and ownership-conservation results. (5) is the audit-monotonicity invariant together with the PCAuth transport contract; every discharged check is paired with its attestation entry and the entries are content-addressed and append-only. ■

This schema is the unified formal statement of "Op is compliance-carrying." A well-typed Op program, starting at the top of the compliance-context lattice, is intended to reach a well-typed value under a monotonically descending compliance context with a complete audit bundle, or terminate at a sanctions-bottom from which no continuation recovers. The full proof is part of the §14 mechanization queue.

7.1 Gas Conservation

Open invariant schema. For every execution trace, the total structural gas consumed is equal to the number of reduction-rule-applications performed on the compiled AST, and the total extensional gas consumed is equal to the sum, over all metered operations, of the declared per-unit rate times the cardinality certificate supplied at dispatch. Abort and compensation states carry the gas meter explicitly, so `OutOfCompensationGas` is a terminal accounting state rather than an unmetered exception.

Proof target. The intended proof is by induction on the reduction relation. Every reduction rule consumes exactly one unit of structural gas (§6.2) and, for a primitive dispatch with extensional metering, exactly the declared extensional charge. The gas meter is monotone decreasing; no reduction rule increases gas. At termination, total consumed equals the difference between initial budget and final balance, which equals the rule-application count on the trace. The bound computed on the compiled AST is a sound upper bound for any legal trace because each rule-applying form in the AST fires at most once per resumption per await site.

7.2 Resource Linearity

Invariant. For every value whose type is `Linear<T>` or `Locked<T, ω, ε>`, the value is consumed exactly once along any reduction path.

Proof sketch. The type system (§4.10) enforces that linear values are introduced by a single expression and that every reduction rule mentioning a linear value either consumes it (the value is absent from the post-state environment) or moves it into a strict sub-continuation where it is consumed exactly once. `Locked<T, ω, ε>` has only two eliminators (`commit_transfer` and `release_lock`), both consuming the value; no other rule mentions `Locked<T, ω, ε>` in an eliminating position. Structural induction on the type derivation gives the invariant on all reachable configurations.

7.3 Ownership Conservation

Invariant. For every state mutation committed in the proof bundle, exactly one principal is recorded as the authorized writer, and this principal holds the credentials that the primitive's lowering rule declares required.

Proof sketch. The primitive dispatch rule (§6.5) records the calling principal's identity as part of the commit entry. The type checker rejects any primitive call whose caller credentials do not carry the authorization required by the primitive's lowering rule. The append-only discipline of the proof bundle (§7.4) prevents later entries from re-attributing ownership of an earlier commit.

7.4 Audit Monotonicity

Invariant (syntactic). The proof bundle `mu` is append-only: every reduction rule either leaves `mu` unchanged or extends it with entries whose content-addressed digests are unique. No reduction rule removes, reorders,

or modifies an existing entry. Compensation entries append; they do not delete the forward entry they invert.

Invariant (semantic). The set of valid certifications derivable from `mu` is non-monotonic in the presence of compensation: a compensation entry for step `s` invalidates downstream consumption of `s`'s certifications. Semantic audit monotonicity is a corollary only for traces without compensation. For traces with compensation, downstream consumers reason over the certification lattice of `mu` (forward commit joined with inverse commit yielding the effective state), not over an append-only set.

Proof sketch. The rules that mutate `mu` are `commit(s)` (§6.5), which appends exactly one entry, suspension `Paused`, which appends one paused-token entry, and compensation, which appends one entry per executed inverse (each a commit of the inverse operation). No rule removes or rewrites. Content-addressing uniqueness follows from the pre-image resistance of the digest function, modulo collisions assumed negligible under standard cryptographic hypotheses.

7.5 Meet-Monotonicity Across Zones

Invariant. Let `T_A : D → TensorValue` and `T_B : D → TensorValue` be compliance tensors produced by Op executions in zones A and B, under pack versions `p_A` and `p_B`, where `D` is the set of compliance domains. Let `phi_{A,B} : D → D` be a corridor translation function witnessing mutual recognition of domain semantics between `(A, p_A)` and `(B, p_B)`. On domains where `phi_{A,B}(d)` is defined and both cells are Applicable, the composed tensor `T_AB(d) = T_A(d) meet T_B(phi_{A,B}(d))` is pointwise no more permissive than either input in the Applicable grade lattice. Mixed applicability cells return `MeetResult` outcomes. Domains outside the corridor's domain of translation fail closed as `Pending` or as a structured missing-translation obstruction; they are not silently treated as `NotApplicable`.

Proof sketch. The Applicable grade lattice is a meet-semilattice with meet the pointwise grade meet (smaller elements more restrictive). Monotonicity of meet gives `T_A(d) meet T_B(phi(d)) ⩽ T_A(d)` and `T_A(d) meet T_B(phi(d)) ⩽ T_B(phi(d))` for all `d` where `phi(d)` is defined and both cells are Applicable. The corridor translation `phi` is an explicit input to cross-zone composition, not implicit: the evaluator rejects or stages a composition lacking a declared translation.

7.6 Information-Flow Security

This section specifies a Denning-style information-flow discipline for Op as a declarative extension of the typing judgment of §7.0 (Denning, 1976; Sabelfeld and Myers, 2003). It states non-interference for a core fragment as an open theorem schema with a proof sketch. The IFC judgment, declassification discipline, and non-interference theorem are not mechanized; §14.14 records the closure route.

Let `(L, ⩽, join, meet, bottom_L, top_L)` be a finite bounded lattice of confidentiality/integrity labels, ordered so that $\ell 1 \leqslant \ell 2$ means information at $\ell 1$ may flow to $\ell 2$. The confidentiality chain

`public = bottom_L ⩽ ... ⩽ top_L = secret` reads upward; integrity is the order-dual, and product lattices combine both. A value of underlying type `T` protected at label ℓ has type `T^`ℓ. Labeling is functorial over type constructors: for `F(T_1, ..., T_n)`, the labeled variant `F(T_1^{`ℓ`_1}, ..., T_n^{`ℓ`_n})^`ℓ is well-formed when $\ell \geqslant$ `join(`ℓ`_1, ..., `ℓ`_n)`, and projections raise the component label to the ambient label.

The IFC judgment

$$\Gamma;\ \mathtt{pc};\ \mathtt{A};\ \Xi \vdash \mathtt{e} : \mathtt{T}\hat{}\ell \ !\ \rho \Rightarrow \Xi'$$

extends the base judgment with a program-counter label `pc` $\in$ `L`, an authority set `A`, and a compliance context Ξ mapping each discharged obligation to the label at which it was established. The `pc` component blocks implicit flows through branching; the authority component is consulted only by explicit declassification.

For observer label ℓ`_o`, write $\lfloor$`v`$\rfloor$`_{`ℓ`_o}` for erasure of value `v`: components labeled $\ell \leqslant \ell$`_o` are preserved and higher components are replaced by $\bullet$. Extend erasure pointwise to environments, proof bundles, gas meters, and final outputs; write `x` $\approx$`_{`ℓ`_o}` `y` when erasures coincide. Categorically, $\lfloor\cdot\rfloor$`_{`ℓ`_o}` is a low-view functor from typed configurations and reductions to observer views; non-interference says this functor is well-defined on equivalence classes generated by multi-step reduction.

Flow-respecting typing rules. The four load-bearing IFC refinements are the guard pc-raising rule and three term rules:

$$\frac{\Gamma;\ \mathtt{pc};\ \mathtt{A};\ \Xi \vdash \mathtt{e\_g} : \mathtt{Bool}\hat{}\{\ell\mathtt{\_g}\} \ !\ \rho\mathtt{\_g} \Rightarrow \Xi\mathtt{\_g} \qquad \Gamma;\ \mathtt{pc\ join}\ \ell\mathtt{\_g};\ \mathtt{A};\ \Xi\mathtt{\_g} \vdash \mathtt{e\_i} : \mathtt{T}\hat{}\ell \ !\ \rho\mathtt{\_i} \Rightarrow \Xi\mathtt{\_i} \quad \mathtt{for\ i} \in \mathtt{\{1,2\}} \qquad \mathtt{pc\ join}\ \ell\mathtt{\_g} \leqslant \ell}{\Gamma;\ \mathtt{pc};\ \mathtt{A};\ \Xi \vdash \mathtt{choose\ e\_g\ then\ e\_1\ else\ e\_2} : \mathtt{T}\hat{}\ell \ !\ (\rho\mathtt{\_g} \cup \rho\mathtt{\_1} \cup \rho\mathtt{\_2}) \Rightarrow (\Xi\mathtt{\_1\ meet}\ \Xi\mathtt{\_2})}\ \text{(T-Choose)}$$

$$\frac{\Gamma;\ \mathtt{pc};\ \mathtt{A};\ \Xi \vdash \varphi : \mathtt{Prop}\hat{}\ell \ !\ \rho \Rightarrow \Xi \qquad \mathtt{pc} \leqslant \ell}{\Gamma;\ \mathtt{pc};\ \mathtt{A};\ \Xi \vdash \mathtt{check}(\varphi) : \mathtt{Unit}\hat{}\ell \ !\ (\rho \cup \mathtt{\{proof\_emit\}}) \Rightarrow \Xi[\varphi \mapsto \ell]}\ \text{(T-Check)}$$

$$\frac{\Gamma;\ \mathtt{pc};\ \mathtt{A};\ \Xi \vdash \mathtt{v} : \mathtt{Locked<}\mathtt{T}\hat{}\ell,\ \omega,\ \varepsilon\mathtt{>} \ !\ \rho \Rightarrow \Xi \qquad \Gamma \vdash \omega : \mathtt{Bridge(z\_s,\ z\_d,}\ \tau\mathtt{\_}\omega\mathtt{,\ clr\_\{z\_d\})} \qquad \tau\mathtt{\_}\omega : \mathtt{L} \to \mathtt{L\ monotone} \qquad \tau\mathtt{\_}\omega(\ell) \leqslant \mathtt{clr\_\{z\_d\}}}{\Gamma;\ \mathtt{pc};\ \mathtt{A};\ \Xi \vdash \mathtt{commit\_transfer}(\omega,\ \mathtt{v}) : \mathtt{T}\hat{}\{\tau\mathtt{\_}\omega(\ell)\} \ !\ (\rho \cup \mathtt{\{fiscal\_transfer,\ proof\_emit\}}) \Rightarrow \Xi}\ \text{(T-Corridor-Commit)}$$

$$\ell' \in \mathtt{L}$$

```
------------------------------------------------------  (T-Sanctions-Bottom)
Γ; pc; A; Ξ ⊢ SanctionsBlocked : void^{ℓ'} ! {sanctions_check} ⇒ Ξ
```

`T-Choose` raises the branch-local pc by the guard label and enforces the sink side-condition on the branch result. `T-Check` records a discharged obligation at the same label at which it was established. `T-Corridor-Commit` requires a monotone label translation carried inside the corridor witness; if both zones share the same IFC lattice, `τ_ω` is the identity and the premise reduces to `ℓ ⩽ clr_{z_d}`. `T-Sanctions-Bottom` types the blocked term with an uninhabited type at every label, so no continuation can recover by retargeting the computation to a different label.

Open theorem schema 7.6.1 (Non-Interference for the Core Fragment). Fix an observer label `ℓ_o`. Let `P` be a well-typed Op program in the IFC core fragment: literals, records, `let`, sequential composition, guarded choice under `T-Choose`, `check(φ)`, and the linear `Locked` eliminators under `T-Corridor-Commit`; excluding `declassify`, `await`, and `sanctions_check`. Suppose initial environments `σ_1, σ_2` agree on every input of label `⩽ ℓ_o`, so that `σ_1 ≈_{ℓ_o} σ_2`. If

```
(P, σ_i, μ_0, G_0, []) -→* (v_i, σ_i', μ_i, G_i, [])    for i ∈ {1,2},
```

then `(v_1, σ_1', μ_1) ≈_{ℓ_o} (v_2, σ_2', μ_2)`. In particular, the low projections of the final value, final store, and proof bundle coincide.

Proof sketch. Use the standard erasure argument. The single-step unwinding lemma states that if `c_1 ≈_{ℓ_o} c_2` and `c_1 -→ c_1'`, then either `c_2` is terminal or there exists `c_2'` with `c_2 -→* c_2'` and `c_1' ≈_{ℓ_o} c_2'`. A guard of label `ℓ_g` not below `ℓ_o` raises the branch-local pc to `pc join ℓ_g`; the sink side-conditions of `T-Choose` and `T-Check` then forbid the branch from writing at any label visible to `ℓ_o`. `T-Check` records a high obligation only at high label, so it cannot alter the low projection of `Ξ` or `μ`. `T-Corridor-Commit` preserves labels under monotone `τ_ω`, so the low observer sees either the same transported payload or no payload. Iterating unwinding along both traces yields final low equivalence; determinism collapses low equivalence to equality of low erasures.

Declassification discipline. The only typed relaxation of Open Theorem Schema 7.6.1 is an explicit declassification operator in the sense of Sabelfeld and Sands (2005):

```
ℓ' ⩽ ℓ    Γ; pc; A; Ξ ⊢ e : T^ℓ ! ρ ⇒ Ξ    Γ ⊢ w : PCAuth(p, ℓ, ℓ', a)    a
↪ ∈ A
-------------------------------------------------------------------------------------------------
↪ (T-Declassify)
Γ; pc; A; Ξ ⊢ declassify_{ℓ → ℓ'}(w, e) : T^{ℓ'} ! (ρ ∪ {proof_emit}) ⇒ Ξ
```

Lawful disclosure therefore has an explicit `PCAuth`: the filler authorizes the declassification, the witness records the purpose `p` and the exact downgrade `ℓ → ℓ'`, and the term is ill-typed without it. In the compiled discretion-hole case, a filled hole may release information only when the filler's attestation authorizes that precise release. The four-dimensional declassification taxonomy of Sabelfeld and Sands (2005) maps onto PCAuth fields, but the full what/who/where/when analysis is not proved here.

Low-erasure-stable certificates. A cardinality or size certificate `n : Nat^{ℓ_c}` is low-erasure-stable for observer `ℓ_o` when `ℓ_c ⩽ ℓ_o`. An extensional-gas charge is low-erasure-stable when it is a function only of such certificates and public worst-case bounds. A high payload may change value, but it may not change the gas charge seen by a low observer unless first declassified.

Open corollary schema 7.6.2 (Timing Non-Interference). Under the hypotheses of Open Theorem Schema 7.6.1, if every extensional-gas charge is low-erasure-stable for `ℓ_o`, then `(G_1, μ_1) ≈_{ℓ_o} (G_2, μ_2)`: the low projections of the gas-and-timing traces coincide.

Proof sketch. Structural gas is fixed by the reduction rules taken, which by unwinding step in low lockstep. Extensional gas is equal on low-erasure-stable certificates by hypothesis. The runtime exposes no wall-clock primitive, so the only timing observation is the metered trace itself.

Cross-zone IFC. Cross-zone flow is governed by `T-Corridor-Commit`. A bridge may preserve labels by identity or translate them by a monotone map `τ_ω`; it may not erase them. A well-typed `Locked<T^ℓ, ω, ε>` is transported only through bridges whose destination clearance dominates the transported label. The receipt recorded in the proof bundle carries the transported label together with `ω` and `ε`, so replay at the destination zone re-checks label preservation rather than trusting the source zone's classifier.

Status. The material in this section is declarative: a judgment extension, theorem schemas, and proof sketches. None of Open Theorem Schema 7.6.1, Open Corollary Schema 7.6.2, or the cross-zone IFC corollary is mechanized in the companion Coq development.

---

8. The Compilation of the Rule Logic to Op

The compilation target from the companion rule logic (Lex) into Op takes a Lex term to an Op contract clause, a Lex predicate to an Op boolean expression, and a Lex rule pack to an Op pack-digest. The compilation is defined on, and total for, the admissible fragment of Lex: the sub-calculus whose terms contain (a) only first-order data (records, variants, lists, options, base literals), (b) no modal operators, (c) no temporal coercions, (d) no unfilled discretion holes (filled holes with value-indexed PCAuth witnesses `w : PCAuth(auth, h, v)` or quorum witnesses `W : PCAuth_k(auth, h, v, k)` are admissible), and (e) pattern-matching whose exhaustiveness is decidable by the host prelude. Every term in this fragment has an Op compilation. No target-side surjectivity claim is made.

8.1 The Compilation Target

Lex terms fall into three classes at the Op boundary:

1. **Flat predicates** over a prelude type. These compile to Op boolean expressions consumable by precondition and postcondition clauses, or by guard expressions in choose blocks.

2. **Defeasible rules** with a finite priority list. These compile to a chain of choose expressions whose guards are the exception guards in descending priority order, with the base body in the else branch.
3. **Compliance fiber bundles.** A Lex rule producing a fiber verdict over a compliance domain compiles to an Op `ensures domains` declaration on the step that produces the verdict's evidence, plus a post-flight predicate that consumes the verdict.

Terms outside the admissible fragment (those exercising the full dependent-type machinery, modal operators, or unfilled discretion holes) have no Op compilation and are rejected at the compilation boundary. Section 14 notes this as an open problem.

8.2 The Compilation Rule

Surface Lex Elaborated Lex Admissible Fragment Gate Op IR Op Bytecode Operation Daemon Proof Bundle parse admit? lower emit load run reject + diagnostic Any upstream rejection short-circuits to the diagnostic terminal; downstream boxes execute only on admission.

Figure 2. Lex to Op compilation pipeline. The admissible-fragment gate is the static boundary that decides whether a Lex term has an Op target. Rejection at any upstream stage short-circuits to the diagnostic terminal; downstream stages never run on inadmissible input.

The compilation `[[.]] : L_adm → Op` is defined by structural induction over the admissible fragment:

- `[[ constant c ]]` = the Op literal representation of `c`.
- `[[ variable x ]]` = the Op reference to the binding of `x`.
- `[[ match e return T with | P_1 ⇒ e_1 | P_2 ⇒ e_2 ]]` = an Op `match` over `[[e]]` with one branch per decidable constructor and a materialized catch-all inhabiting `T`. The admissible fragment's decidable-exhaustiveness clause guarantees the catch-all is unreachable on any admissible context; the branch is present as a total-function closure, not as a semantic case. Verdict-valued matches use the canonical `NonCompliant("pattern_unmatched")` catch-all; scalar-valued matches use the typed inert inhabitant required by the Op checker.
- `[[ defeasible b priority p_0 unless (g_1, b_1, p_1) | (g_2, b_2, p_2) | ... ]]` with exceptions sorted by (p_i descending, source-position ascending), a total order the admissibility predicate of the rule logic requires, compiles to `choose { when [[g_1]] → [[b_1]]; when [[g_2]] → [[b_2]]; ...; else → [[b]] }`. Admissibility rejects defeasible rules whose exception list fails to totally order under (priority, source-position); the compilation of a non-admissible defeasible rule is undefined and rejected at the compilation boundary.
- `[[ sanctions_query e ]]` = the Op step `sanctions.check` applied to `[[e]]`, with the `sanctions_check` effect declared and the distinguished fail-closed semantics for applicable `NonCompliant` verdicts. The companion rule logic names the effect `sanctions_query` at the predicate layer, reflecting its predicate-query idiom; Op names it `sanctions_check` at the

operational layer, reflecting its dispatch idiom. Both names refer to the same semantic effect and compile to the same runtime dispatch.

- `[[ fill(hole, value, witness) ]]` = the Op literal `[[value]]`, with the PCAuth witness persisted as a predicate-attached attestation in the proof bundle. The attached-attestation discipline instantiates Necula's (1997) proof-carrying code pattern: the compiled value ships with a machine-checkable record of the authority under which the discretion hole was filled, and a receiving zone validates the attestation independently of trusting the source zone's evaluator. The target receiving-zone verifier runs `VerifyPCAuth(witness, hole, value, t_use)`: it re-checks every Ed25519 signature over the exact tuple `(signer, hole, value)`, verifies that the attached witness satisfies the hole's quorum threshold, verifies the linked timestamp anchor for `(hole, value)`, rejects any credential or delegation link revoked at or before `t_use`, and rejects any delegation chain whose realized length exceeds its explicit depth bound. A revocation after `t_use` taints the proof-bundle entry for downstream re-evaluation without retroactively fabricating authority. If the witness fails validation, the compiled site emits `AuthorizationBlocked` regardless of the literal `[[value]]`; the terminal is fail-closed and non-overridable like `SanctionsBlocked`, and it records an authorization failure rather than a sanctions verdict. The current public mechanized compilation-soundness fill case treats the witness transport as an uninterpreted authority/digest/timestamp payload; delegated revocation, quorum, and cross-zone bridge validation are verifier obligations rather than closed mechanized compilation theorems. The admissibility gate enforces the precondition that a filled hole whose type contains `SanctionsVerdict` as a subterm ships the corresponding PCAuth witness; a fill without the witness is inadmissible and has no Op compilation.

8.3 PCAuth Verifier and Transport

The compilation rule above states that a filled discretion hole ships with a PCAuth witness in the proof bundle. This subsection fixes the transport form and verifier obligations of that witness at every receiving Op evaluator. Write the transported witness as `w : PCAuth(auth, h, v)`, where `h` is the hole descriptor and `v` the supplied value. In this transport layer, `AuthorityChain` is the credential path from a signer to the named authority, `ScopeWitness(h.scope, v)` is the evidence that the supplied value respects the hole's scope, `LinkedTimestamp` is the bulletin-board inclusion proof that anchors the witness to a public root in the sense of Haber and Stornetta (1991), and `MutualRecognitionTreaty(A, B, scope)` is the typed certificate that zone `B` recognizes zone `A`'s authority over the stated scope.

Target verifier duty. Given a PCAuth witness embedded in a proof bundle, the Op verifier accepts it only if all of the following checks succeed. This is the target transport contract; the currently mechanized fill-transport case covers only the uninterpreted witness fields needed for value-preservation.

1. **Signature validity.** `Ed25519-verify(signers, (h, v), sigs)` holds.
2. **Delegation chain.** For each signer, the associated `AuthorityChain` up to `auth` is well-formed and has depth at most `max_depth`.

3. **Scope matches hole.** `ScopeWitness(h.scope, v)` checks.
4. **Timestamp anchoring.** The witness's `LinkedTimestamp` matches a known public-bulletin root.
5. **Non-revocation.** At evaluation time the verifier consults either the current revocation list or an oracle-signed non-revocation statement covering every signer credential used by the witness.
6. **Quorum.** `quorum_ok` holds if and only if the set of distinct signers has cardinality at least `k`.

Failure of any clause rejects the witness and restores the same fail-closed behavior the compilation rule assigns to an invalid fill.

Gas cost. PCAuth validation is a refinement-typed Op operation. For a witness carrying `n` signatures, quorum threshold `k`, and delegation-depth bound `d`, the gas schedule is

```
gas_pcauth(n, k, d) = c_sig * n + c_revoke + c_scope + c_anchor.
```

The indices `k` and `d` remain part of the operation's refinement type because quorum and delegation depth are checked at validation time, but the current cost commitment charges linearly only in the number of signatures plus three named constant terms. The numeric values of `c_sig`, `c_revoke`, `c_scope`, and `c_anchor` are deferred to a future tuning pass.

Caching protocol. Each verifier maintains a validated-PCAuth cache keyed by the hash of `(h, v, sigs)`. The cache stores the validation verdict, the signer set, and the revocation dependencies needed for invalidation. The cache entry TTL is `min(signer credential expiry, bulletin-board root age)`. On a revocation event, the verifier flushes every cache entry whose stored signer set contains the revoked credential. An implementation therefore maintains the cache together with a reverse index from signer credential to cached witness keys.

Expiration enforcement. The transport form of PCAuth carries `expires_at : Time_0`. The verifier rejects the witness whenever `now > expires_at`. Expiration is checked after timestamp anchoring and before the witness is admitted to the cache.

Bulk verify primitive. Op exposes `verify_batch(witnesses : Vec PCAuth) : Vec BoolVerdict`. The primitive runs batch Ed25519 verification following Boneh, Drijvers, and Neven (2018), giving amortized cost `O(n + log n)` over the signature-validation component. Batch verification does not waive the remaining per-witness checks: delegation depth, scope, anchor, expiration, revocation, and quorum are still evaluated entrywise before the corresponding `BoolVerdict` is returned.

Cross-zone PCAuth bridge. A PCAuth witness issued by authority `auth_A` in zone `A` is accepted in zone `B` if and only if a `MutualRecognitionTreaty(A, B, scope)` exists for the relevant scope. Op types the bridge as `Bridge_MRT : PCAuth_A → PCAuth_B`. The base case is a direct treaty witness from `A` to `B`; a corridor may also carry a composed bridge chain `A = Z_0 → Z_1 → ... → Z_m = B` of treaty witnesses. The verifier checks every link in that chain, rejects if any link fails scope or signature validation, and rejects if no bridge chain terminates at the receiving zone.

Multi-signer transport. The transport form of a quorum PCAuth witness carries `Vec_n Ed25519Sig` together with the aligned signer vector and delegation chains. The bytecode carrier uses a compact deterministic serialization, either canonical CBOR or Protobuf with a fixed field order declared by the pack digest, so that every zone hashes the same byte sequence for caching and replay. A bridge witness that transports such a quorum witness across zones must also translate the quorum rule: besides the treaty chain it carries the proof that the source-zone quorum requirement discharges the destination-zone requirement for the same scope. A bridge that cannot justify this quorum translation is ill-typed and the transported witness is rejected.

8.4 Preservation Theorem (Verdict Soundness)

Theorem (verdict soundness). Assume Lex evaluation is deterministic on the admissible fragment (pattern-match and defeasible-priority reductions fire in source order; no ambient nondeterminism). Let `r` be a Lex rule in the admissible fragment, `[[ r ]]` its Op compilation, and `ctx` a valid context. If the Lex evaluation of `r` on `ctx` produces a verdict `v`, then the Op execution of `[[ r ]]` on `ctx` produces a trace ending in a proof-bundle entry with verdict `v`. Mechanized for all nine compilation cases in Rocq 9.1.1 in a development with two shared-model parameters (the deterministic sanctions host primitive and the prelude lookup); the scoped range condition on the host oracle is a local hypothesis of the sanity example, not a file-level axiom.

Define the observable label of an Op reduction as the pair (verdict-introducing-step?, committed verdict if so). Structural-gas ticks, proof-bundle extensions that do not introduce a verdict, and effect-row growth are internal (τ-labelled) transitions. The theorem above gives the Lex-to-Op simulation direction: every observable Lex verdict step is matched by a finite Op execution with the same observable verdict. The converse matching direction is the completeness obligation of §8.6 and is not claimed here.

Proof sketch. By induction on the Lex reduction derivation. Leaves: Lex literals and variables correspond to Op literals and references. Match: the Lex pattern-match reduction produces a branch value; Op `choose` produces the same branch value under the same guard. Defeasible: the Lex `defeat` operator reduces to the highest-priority satisfied exception's body; Op `choose` with guards in descending priority reduces to the same body. Sanctions: Lex `sanctions-dominance(p)` produces bottom; Op `SanctionsBlocked` halts the program; both yield a non-permissive outcome that propagates through any downstream composition. Filled discretion holes: Lex `fill(h, v, w)` reduces to the literal `v` carrying the value-indexed PCAuth witness `w : PCAuth(auth, h, v)`; Op compiles to the literal `[[v]]` with `w` persisted as a predicate-attached attestation. The mechanized fill case preserves the value while transporting the witness payload. The stronger receiving-zone checks for signature messages, quorum threshold, timestamp anchoring, non-revocation, and delegation-depth bounds are part of the verifier contract above and are outside the closed compilation-soundness proof. The attestation append extends `mu` with a τ-labelled step carrying no verdict introduction; the step's observable label matches the Lex reduction's.

Scope of the claim. Verdict soundness holds for the admissible fragment. If an Op-specific terminal configuration (`OutOfStructuralGas`, `Timeout`, compensation-pass abort) occurs before a verdict entry, the

soundness theorem has no conclusion for that run; those terminal behaviours are part of the open completeness statement, not a vacuous branch of the theorem. For the full Lex calculus the compilation is partial: modal operators, temporal coercions, and unfilled discretion holes have no Op target. §14 enumerates these gaps.

8.5 Coq Mechanization Status

This section reports the Rocq Prover 9.1.1 mechanization state of every theorem stated in this paper. The report is stratified by how directly each Coq statement ranges over the Op language of §6: theorems whose Coq statement ranges over the full Op calculus or an identified sub-language of the Lex→Op compilation target are listed under "Op-facing"; theorems currently Qed-closed only over concrete toy fragments that share none of Op's defining constructs are listed separately; parametric-interface modules are listed with their declared Axioms. The complete Parameter and Axiom inventory follows at the end of the section. No `Admitted.` theorems remain in the repository; no classical axioms are imported or used.

Qed over Op-facing semantics The following results are mechanized over terms, values, and state models that match the Op language or the Lex→Op compilation target of §8.2:

- **Verdict-preservation** (`op/formal/coq/CompilationSoundness.v`). Nine of nine compilation cases close with `Qed`: `verdict_preservation_const`, `verdict_preservation_sanctions`, `verdict_preservation_var`, `verdict_preservation_const_record`, `verdict_preservation_const_list` `verdict_preservation_const_variant`, `verdict_preservation_match`, `verdict_preservation_defeasi` and `verdict_preservation_fill`. The mechanized source and target are restricted to the first-order scalar, record, list, variant, match, defeasible, and sanctions-dominance cases of the admissible fragment. These case theorems support the Lex-to-Op verdict-soundness direction stated in §8.4. The file declares two file-level `Parameter`s (`host_sanctions`, `prelude`) modelling the deterministic host oracle and the compile-time prelude lookup; `Print Assumptions` on the case theorems reports exactly those two Parameters. A scoped `Hypothesis host_sanctions_range` inside `Section SanctionsRangeSanity` bounds the oracle image at `{Compliant, SanctionsBlocked}`, but it is not an Axiom at file scope: any concrete host binding discharges it with a proof.
- **Admissible-fragment verdict agreement** (`op/formal/coq/LexOpAdequacy.v`, `OpMetaTheory.v`). The current Coq development also Qed-closes a bidirectional verdict-agreement theorem over the deliberately restricted admissible-fragment verdict alphabet; the file now exposes descriptive aliases `lex_op_finite_verdict_agreement` and `lex_op_finite_verdict_agreement_bisim` for the historical theorem names `lex_op_adequacy` and `lex_op_adequacy_bisim`. This is useful as a finite-model check of the nine compilation clauses, but it is not the full paper-level adequacy theorem: it does not prove compliance-context full abstraction, it does not cover the full Lex calculus, and it does not discharge Op-specific terminal behaviours. Directions (b) and (c) of §8.6 remain open; mechanizing them requires coinduction for the reverse operational direction and formalizing the compliance-carrying judgment for context adequacy.

- **Bilateral session protocol safety** (`op/formal/coq/SessionCorridor.v` + `op/formal/coq/MPSTProjection.v`). Six-message alphabet `{MsgTensorRequest, MsgLockedTensor, MsgVerdictI, MsgVerdictR, MsgDecision, MsgAck}` with I5_Ack / R5_Ack states; `deadlock_freedom`, `session_safety`, `no_mixed_decision`, `no_mixed_ack`, `ack_rung_erasure`, `ack_rung_preserves_safety` all Qed. Concrete `G_corridor_ack` with six labels pinned (`lbl_TensorRequest=0` through `lbl_AbortAck=7`) and duality theorems `G_corridor_ack_duality`, `G_corridor_ack_duality_sym`, `G_corridor_no_ack_duality`, `G_corridor_prefix_shared` all Qed. *Disclosure:* the message payload type is uninterpreted. Both `MPSTProjection.v` and `SessionDuality.v` declare `Parameter payload : Type.`, so the Qed-closed safety and duality theorems are proved over the message-ordering skeleton of the bilateral protocol, not over an instantiated indexed-typestate payload. Instantiating `payload` to the concrete `Locked<T,ω,ε>/Signed<V,ω,ε>/Verified<ω,ε>` family of §6.11 is open work listed in §14.
- **BSC invariants I1, I2, I3** (`op/formal/coq/BSCInvariants.v`). Corridor history `kappa` is modelled as a four-component record (locks, signs, verifieds, blames). Qed-closed: `invariant_I1`, `invariant_I2`, `invariant_I3` as the three invariants; `op_{lock,sign,verify,blame,timeout}_preserves_bs` one per rule; `verified_durable_op_{lock,sign,verify,blame,timeout}` for Verified-entry durability under non-timeout operations; `equivocation_detected_by_I2` and `blame_condition_violates_I2` as the equivocation-extraction witness used by the SJN accountable-disagreement obligation; `bsc_invariants_all_reachable` trace-level lift. *Disclosure:* the history `kappa` is an abstract four-component record. The theorem statements depend on the declarative lock/sign/verify/blame/timeout state machine over abstract entries, not on the concrete operational semantics of the Op expression language.
- **Byte-level wire format and independent verifier** (`op/formal/coq/WireFormatVerifier.v`). Five-byte canonical encoding `[version, tag, entity_byte, op_byte, verdict_byte]` with six rejection classes. Total reference verifier `verify : list byte → verifier_result`. Correctness Qed: `encode_verify_roundtrip`, `encode_verify_accepts`, `verify_accepts_canonical`, `verify_reject_non_canonical`, `verify_deterministic`, `encode_injective`, `byte_content_addressable`. Conformance suite: six accept + six reject tests, all Qed with `reflexivity`.
- **Lex⊠Op verdict compatibility** (`op/formal/coq/LexVerdictEmbedding.v`). This module proves a Qed-closed embedding from the Lex five-element verdict chain into the finite operational carrier used by the current Op Coq context model, including injectivity, rank monotonicity, meet preservation on the embedded fragment, and a partial inverse on the image. This is a compatibility result for the simplified verdict carrier in the mechanization. It is not the full SJN tensor theorem: mixed applicability, provenance, and route-coherent tensor composition live in the two-axis compliance tensor and remain separate proof obligations at the SJN/Op boundary.
- **Canonical-encoding content-addressability** (`op/formal/coq/CanonicalEncoding.v`), **compliance-context meet-semilattice** (`op/formal/coq/ComplianceContext.v`), and **effect-**

row lattice (`op/formal/coq/EffectRow.v`). Each closes its module-local Qed obligations over the stated lattice/encoding object.

Qed only over concrete toy fragments (the five deferred theorems) Termination, progress, subject reduction, effect monotonicity, and parallel confluence are stated in the paper over the Op language of §6. None of them is yet mechanized over Op proper. The repository instead carries a parametric abstract interface `op/formal/coq/OpPaperTargetsModuleType.v` whose concrete witness `op/formal/coq/OpPaperTargetsInstance.v` binds the interface to the following toy fragments. Inhabitation of the abstract Module Type by a toy instance does not establish the theorem for Op.

- **Progress and subject reduction** (`op/formal/coq/OpProgressSubject.v`): Qed over a simply-typed lambda calculus with a shared prelude. The mechanized value and expression forms are `E_Const`, `E_Var`, `E_Lam`, `E_App`, `E_Let`; the typing judgment has no Op-specific shape (no κ history threading, no effect row, no linearity, no compensation, no await, no par, no choose).
- **Termination** (`op/formal/coq/OpConcreteAST.v` instantiated in `OpPaperTargetsInstance.v`): `concrete_termination` closed over a 9-constructor gasified AST (constants, variables, let, lambda, application, sanctions primitive, halted form) by well-founded induction on structural gas. The file declares `Axiom gas_decreases : ∀ c c', step c c' → gas c' < gas c.` as a module-level axiom; structural normalization follows from the axiom, not from a semantic termination argument over Op's operational semantics.
- **Effect monotonicity** (`op/formal/coq/OpEffectMonotonicity.v`): `op_effect_monotonicity_empty_start` closed over a synthetic two-slot effect-tracking machine with an abstract `runion` operation. The machine records two counter slots and a compensation stack of effect rows; its ad-hoc reduction rules `step_a`, `step_b` increment the counters. Not an Op-operational-semantics statement.
- **Parallel confluence** (`op/formal/coq/OpEffectMonotonicity.v`): `par_confluence_diamond` closed as a local diamond over the same two-slot counter machine. Not a confluence proof for Op's `par` combinator of §6.6.

The full mechanization of these five theorems over the operational semantics of Op, along with the κ threading in the typing judgment, the `Linear<T>` consumption discipline, the indexed `Locked`/`Signed`/`Verified` state transitions, and the scoped-compensation and typed-suspension reduction rules, is open work enumerated in §14. The toy-instance Qed-closure establishes only that the module-type signatures are inhabited; it is not a substantive proof about Op.

Parametric modules with declared Axioms The following modules state theorems as parametric interfaces. Concrete instantiation of their Parameters and Axioms against Op's real operational semantics is part of the same open-work queue:

- `op/formal/coq/BundleAppendOnly.v`: `Axiom bundle_step_prefix : ∀ c c', step c c' → Prefix (bundle c) (bundle c').` over an abstract step relation.

- `op/formal/coq/CorridorMonotone.v`: `Axiom rule_step_extends : ∀ v κ κ', rule_step κ κ' → κ ⊆ κ'.` over an abstract rule-step relation.
- `op/formal/coq/GasTermination.v`: `Axiom gas_decreases : ∀ c c', step c c' → gas c' < gas c.` over a parametric `GasStepSemantics` module.
- `op/formal/coq/UpToTauCompatibility.v`: `Axiom step_deterministic` and `Axiom tau_decreases` over a parametric up-to-τ framework.
- `op/formal/coq/HeteroBisimulation.v`: `Axiom pi_tau : pi A.tau = B.tau.` plus three `Parameter`s `T_Op`, `pi_inv`, `compile` over a parametric bisimulation framework.

Disclosure summary File- or module-level **`Parameter`**s: `CompilationSoundness.host_sanctions`, `CompilationSoundness.prelude`, `MPSTProjection.payload`, `SessionDuality.payload`, and three module Parameters in `HeteroBisimulation` (`T_Op`, `pi_inv`, `compile`).

File- or module-level **`Axiom`**s: `BundleAppendOnly.bundle_step_prefix`, `CorridorMonotone.rule_step_extends`, `GasTermination.gas_decreases`, `OpConcreteAST.gas_decreases`, `UpToTauCompatibility.step_deterministic`, `UpToTauCompatibility.tau_decreases`, `HeteroBisimulation.pi_tau`. A scoped `Hypothesis host_sanctions_range` exists inside `CompilationSoundness.Section SanctionsRangeSanity` but is not a file-level Axiom.

`Admitted.`: none in any of the mechanization files.

Open paper-level mechanization: (i) the five deferred theorems over Op proper, termination, progress, subject reduction, effect monotonicity, parallel confluence, with κ threading, linearity, indexed typestates, compensation, suspension, par, and choose semantics (§14); (ii) the payload instantiation for the bilateral session protocol (§14); (iii) concrete instantiation of the five parametric modules against Op's operational semantics (§14); (iv) adequacy direction (b) via mechanized coinduction up-to-τ (§14); (v) adequacy direction (c) via the mechanized compliance-carrying judgment (§14).

8.6 Adequacy of Op for the Admissible Fragment

Verdict soundness (§8.4) is a stepwise forward correspondence between a Lex reduction and its Op compilation at the level of individual verdict-introducing steps. It is the scaffolding for a stronger adequacy target: that Op is the right execution substrate for the admissible fragment of Lex, in the operational-correspondence sense that Plotkin (1977, "LCF Considered as a Programming Language") introduced for LCF and Milner (1975, "Fully Abstract Semantic Models") refined into the full-abstraction discipline. The full adequacy theorem remains the central metatheoretic target for treating Lex and Op as a two-layer system rather than two unrelated languages with a stapled compilation pass.

Setup. Let `L_adm` $\subseteq$ `Lex` be the admissible fragment (first-order data, no modals, no temporal coercions, filled holes carrying PCAuth witnesses, decidably-exhaustive matches). Let `[[·]] : L_adm → Op` be the compilation of §8.2. Write `Lex-eval(M, ctx)` $\Downarrow$ `v` for "the Lex reduction of `M` under compliance context `ctx` terminates with verdict `v`," and `Op-run([[M]], ctx)` $\Downarrow$ `v'` for "the Op reduction

terminates with `v'` recorded in the proof bundle." Write `final_context(M, ctx)` for the compliance context Lex produces at termination, and `κ_terminal` for the Op compliance-context component of the terminal configuration. Write `v ≡_verdict v'` for verdict-level equivalence on the shared prelude vocabulary `{Compliant, NonCompliant, Pending, SanctionsBlocked}`. Write `T_Op = {OutOfStructuralGas, Timeout, SanctionsBlocked, AuthorizationBlocked}` for the distinguished set of Op-specific operational terminals: configurations with no Lex preimage.

Theorem schema (Adequacy of Op for L_adm). For every `M ∈ L_adm` and every well-typed compliance context `ctx`, the intended adequacy statement has three directions:

(a) **Soundness.** `Lex-eval(M, ctx) ⇓ v` implies `Op-run([[M]], ctx) ⇓ v'` with `v ≡_verdict v'`.

(b) **Completeness, modulo operational terminals.** `Op-run([[M]], ctx) ⇓ v'` with `v' ∉ T_Op` implies `Lex-eval(M, ctx) ⇓ v` with `v ≡_verdict v'`.

(c) **Full abstraction on compliance contexts.** `final_context(M, ctx) = κ_terminal([[M]], ctx)` under the compliance-carrying judgment `Γ | κ ⊢ e : T ! ε ⇒ κ'`.

The three directions correspond to the three classical adequacy obligations for a compiler into a typed bytecode: Plotkin-style soundness (every source-level evaluation has a target-level witness), completeness up to target-specific termination modes (Leroy 2009 for CompCert's treatment of observable behaviors), and full abstraction on the object of observation, here, the compliance context, as refined by Ahmed (2006, "Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types") to carry a typed resource component through the correspondence. Direction (a) is the theorem presently supported by the stepwise verdict-preservation development; directions (b) and (c) are open obligations, as disclosed in §8.5 and §14.

Proof sketch of (a) Soundness. By structural induction on the Lex reduction derivation `M →* v`. The inductive hypothesis is the single-step verdict-preservation statement of §8.4, extended from "a verdict-introducing step matches a τ-chain then a verdict-introducing Op step" to "a full Lex reduction matches a sequence of Op τ-chains, each terminated by the Op step that introduces the corresponding Lex verdict." Leaves (constants, variables via the shared prelude lookup, filled discretion holes) close by the verdict-preservation base cases (`verdict_preservation_const`, `verdict_preservation_var`, `verdict_preservation_fill` in `CompilationSoundness.v`). Inductive cases (match, defeasible, sanctions, record/list/variant composition) close by the verdict-preservation step cases, composed through the τ-preserving context closure: if the subterm reduction preserves verdicts, the surrounding record/list/variant/match context also preserves verdicts because each compositional `[[·]]` clause is a structural homomorphism that τ-pads internally. The Plotkin 1977 LCF development is the template. Nine of nine cases are mechanized in `CompilationSoundness.v` with `Qed`.

Open proof obligation (b) Completeness up to operational terminals. The intended proof is by coinduction on the Op reduction sequence, projected onto the observable-label alphabet of §8.4 (verdict-

introducing step, committed verdict). The proof technique is weak bisimulation up-to-τ in the style of Milner's full-abstraction program (1975, 1977) and the CCS-tradition up-to-τ coinduction of Sangiorgi (1998, "On the Bisimulation Proof Method"). This direction is not closed in the public mechanization.

Open proof obligation (c) Full abstraction on compliance contexts. The intended proof is by structural induction on the Lex typing derivation using the compliance-carrying judgment `Γ | κ ⊢ e : T ! ε ⇒ κ'` of §7. The CompCert development (Leroy 2009; Leroy 2006) is the closest methodological analogue: preservation of an observable state component across a compiler pass under a simulation relation indexed by source-side typing. This direction is a target theorem, not a closed result in this paper.

Compositionality invariant. A proof of direction (c) would make explicit use of the *compositionality invariant*: for every admissible-fragment constructor `C`, if each immediate sub-derivation produces compliance-context update $\kappa \longmapsto \kappa\_i'$ on the Lex side and the same update $\kappa \longmapsto \kappa\_i'$ on the Op side, then the surrounding term `C[...]` should produce a single update $\kappa \longmapsto$ `meet_i` κ_i' on both sides, where `meet_i` denotes the iterated compliance-context meet of §4.11. The Lex side computes this by evaluating each sub-derivation and meeting the results; the Op side computes this by lowering each sub-derivation into an Op step whose commit extends the proof bundle with the sub-derivation's contribution, then taking the pointwise meet at the terminal proof-bundle entry. Closing this invariant for the full compliance-carrying judgment is part of the open direction (c).

Contextual-equivalence target for (c). The (c) direction has a concrete contextual-equivalence target. Let `M_1` and `M_2` be two Lex terms in `L_adm` whose compliance-context updates under every `ctx` are pointwise equal: `final_context(M_1, ctx) = final_context(M_2, ctx)` for every valid `ctx`. Direction (c) would imply κ`_terminal([[M_1]], ctx) =` κ`_terminal([[M_2]], ctx)` for every `ctx`: the two Op compilations would be contextually equivalent on the compliance-context observable. Concretely, consider a defeasible KYC rule

- `M_1 = defeasible(Compliant, [suspicious ⇒ Pending])` and
- `M_2 = match customer with suspicious(_) ⇒ Pending | _ ⇒ Compliant`.

`M_1` and `M_2` produce the same Lex verdict and the same compliance-context update on every `ctx` (the admissible-fragment reduction discipline evaluates the defeasible exception against the same predicate the match arm guards against). The open direction (c) would state that their Op compilations agree on the terminal compliance context: κ`_terminal([[M_1]], ctx) =` κ`_terminal([[M_2]], ctx)` for every `ctx`. This is the target behavioural equation on the Op side derived from a source-side equivalence; the direction is Lex ⇒ Op, and this is the intended sense of "full abstraction on the compliance-context observable."

Carve-out remark on operational terminals. The completeness direction (b) excludes `T_Op` by necessity, not by oversight. Each excluded terminal is a deliberate design choice:

- `OutOfStructuralGas` is an Op-level metering terminal. Lex is fuel-bounded by admissibility (§8.2), a static finite-budget condition; it does not meter individual reduction steps. An Op program that exhausts structural gas on a particular input corresponds to a Lex term whose reduction on that input is finite in principle but exceeds the Op budget. Op's choice to fail closed at the budget boundary is a cross-zone replay requirement.
- `Timeout` is the typed-suspension terminal. Op deadlines are callback-event-count offsets; Lex has no temporal coercion in the admissible fragment. An Op timeout is a liveness failure of the callback substrate, not a semantic outcome of the Lex rule.
- `AuthorizationBlocked` arises at a receiving zone's PCAuth witness re-validation (§8.3, `fill` case). It is a fail-closed authorization terminal with no Lex verdict preimage. `SanctionsBlocked`, by contrast, is reserved for sanctions non-compliance or the corresponding Lex bottom-at-sanctions reduction.

The intended completeness theorem must prove that outside `T_Op`, every relevant Op terminal has a Lex preimage; within `T_Op`, the terminals are Op-specific by design. The carve-out is therefore a proposed proof boundary, not a closed result.

Mechanization status. Direction (a) is mechanized case-by-case in `CompilationSoundness.v`; the nine compilation cases are the nine cases of adequacy direction (a). Directions (b) and (c) remain open. The preceding paragraphs are theorem schemas and proof strategies: direction (b) would require a coinductive up-to-τ argument in Rocq or Lean, naturally via the Paco library, tracking Sangiorgi's up-to-τ discipline; direction (c) requires formalizing the compliance-carrying judgment in the same development.

8.7 The Boundary is Static

Op does not re-interpret Lex predicates at runtime. Compilation happens at program authoring time, not at execution time. This keeps Op's runtime behavior deterministic: a Lex term's Op compilation is a fixed, content-addressed artifact, and every receiving zone that re-evaluates the Op program does so against the same compiled artifact. If a Lex rule changes, the pack version changes, compilation produces a different Op program, and that different program is separately content-addressed. Cross-zone replay is safe across pack updates by version-pinning.

---

9. Bisimulation Theory

The open adequacy directions of §8.6 require a weak-bisimulation argument. This section makes the intended relation explicit in the CCS/coinduction line of Milner (1980), Park (1981), and Sangiorgi (2011), and names the up-to proof method of Pous and Sangiorgi (2007) that would keep the Op congruence proofs finite. Except where a result is explicitly marked as a definition or standard order-theoretic proposition, this section states proof obligations for the Op calculus rather than closed mechanized theorems.

9.1 Weak Bisimulation and the Silent-Step Boundary

Let `Conf_Op` be the set of well-typed Op configurations of §6, and let `c → c'` be the one-step reduction relation. Equip → with labels `a ∈ Obs_Op ∪ {τ}` by `c -a→ c'` iff `c → c'` and:

1. `a = τ` when the step is a structural-gas tick whose only semantic effect is the mandatory decrement of `G.structural`.
2. `a = τ` when `mu' = mu ++ delta` and every entry of `delta` is a proof-bundle extension with no committed verdict.
3. `a = τ` when the step only grows the effect row recorded at the current node.
4. `a = obs(c, c')` otherwise.

All other one-step reductions are observable. Write `c ⇒_tau c'` for the reflexive-transitive closure of `-tau→`, and for `a ≠ τ` write `c ⇒_a c'` for `c ⇒_tau · -a→ · ⇒_tau c'`.

Definition (weak bisimulation). A relation `R ⊆ Conf_Op × Conf_Op` is a weak bisimulation when, whenever `c R d`: (i) every `c -tau→ c'` is matched by some `d ⇒_tau d'` with `c' R d'`; (ii) every `c -a→ c'` with `a ≠ τ` is matched by some `d ⇒_a d'` with `c' R d'`; and (iii) symmetrically every transition from `d` is matched from `c`.

Intuition. Weak bisimulation forgets Op's administrative bookkeeping but keeps every step that changes the user-visible proof, terminal outcome, or corridor dialogue.

9.2 Coinduction and Coinductive Witnesses

Let `Rels = P(Conf_Op × Conf_Op)` and define `F : Rels → Rels` by `(c, d) ∈ F(R)` iff the three matching clauses of §9.1 hold with `R` in the recursive positions. `F` is monotone under inclusion.

Proposition (greatest-fixed-point characterization). Weak bisimilarity `≈_w` is the greatest fixed point `νF`. Equivalently, `c ≈_w d` iff there exists a relation `R` such that `(c, d) ∈ R` and `R ⊆ F(R)`.

Proof. If `R ⊆ S`, every matching family witnessing `(c, d) ∈ F(R)` also witnesses `(c, d) ∈ F(S)`, so `F` is monotone. Park's coinduction theorem therefore yields a greatest post-fixed point `νF`, and by construction its elements are exactly the pairs related by some weak bisimulation.

For Rocq, the witness is encoded coinductively:

```
CoInductive wbisim : cfg → cfg → Prop :=
| wbisim_intro : forall c d,
    (forall a c', step c a c' →
       exists d', weak_step d a d' /\ wbisim c' d') →
    (forall a d', step d a d' →
       exists c', weak_step c a c' /\ wbisim c' d') →
    wbisim c d.
```

The corecursion principle is the standard `CoFix` rule: to prove `wbisim c d`, it suffices to build a guarded `CoFix` term whose recursive calls appear only underneath `wbisim_intro`; each recursive call is precisely the residual obligation `(c', d') ∈ R` for some post-fixed point `R ⊆ F(R)`.

Intuition. A coinductive witness is a potentially infinite matching strategy: every time one side steps, the witness produces the other side's matching weak step and recurs on the residual pair.

9.3 Up-To Techniques

Let `≈_w` denote `vF`. For a relation `R`, define `bisim(R) = ≈_w ∘ R ∘ ≈_w`, `ctx(R) = { (K[c], K[d]) | K ∈ Ctx_core, (c, d) ∈ R }`, and `tr(R) = R+`, where `Ctx_core` is the grammar of core evaluation contexts generated by `let`, `par`, `choose`, and `compensate`. A relation progresses up to `f` when `R ⊆ F(f(R))`.

Proof obligation (sound up-to techniques for Op). For the Op labelled reduction, up-to-bisimilarity `bisim`, up-to-context `ctx`, and up-to-transitivity `tr` should be sound in the sense of Pous and Sangiorgi: if `R ⊆ F(bisim(R))`, or `R ⊆ F(ctx(R))`, or `R ⊆ F(tr(R))`, then `R ⊆ ≈_w`. Finite joins and compositions of these three techniques should be sound as well.

Proof sketch. Up to bisimilarity: composition with `≈_w` preserves progress because `≈_w` is itself a bisimulation. Up to context: the reduction relation is closed under the evaluation-context grammar of §6.3, and the three τ-classes above remain τ after context injection, so a match inside a core context lifts to a match of the surrounding terms. Up to transitivity: weak bisimilarity is transitive, so a finite `R`-chain can be collapsed to a single `≈_w` step. The Pous-Sangiorgi compatibility criterion then yields soundness of the generated closure.

Intuition. The up-to rules let a proof ignore already-solved bisimilar subterms, peel away a shared surrounding context, and compress short chains of intermediate states.

9.4 Congruence for the Core Combinators

Proof obligation (congruence for `let`, `par`, `choose`, and `compensate`). If `c ≈_w d`, then for every well-typed core context `K ∈ Ctx_core`, `K[c] ≈_w K[d]`. In particular:

- `let x : T = c in e ≈_w let x : T = d in e`
- `par { a = c; b = e_b } ≈_w par { a = d; b = e_b }`
- `par { a = v_a; b = c } ≈_w par { a = v_a; b = d }`
- `choose { when c → b_1; ... } ≈_w choose { when d → b_1; ... }`
- `step s { c } compensate { inverse } ≈_w step s { d } compensate { inverse }`

Proof sketch. Let `R = { (K[c], K[d]) | c ≈_w d, K ∈ Ctx_core }`. Because reduction is closed under evaluation contexts, every transition of `K[c]` is either a transition of the hole or a context-managed transition that is identical on both sides. The only nontrivial cases are `par` and `choose`, where

the active branch or guard must be matched inside the surrounding syntax. Up-to-context discharges exactly those cases, so `R ⊆ F(ctx(R))`; soundness of up-to-context then yields `R ⊆ ≈_w`.

Intuition. Replacing a subprogram by a weakly bisimilar one does not change the behaviour of any of Op's core combinators.

9.5 Cross-Zone Commit as Local-Global Bisimulation

Let a global corridor state be `g = (ell_A, ell_B, M, kappa)`, where `ell_i` is zone `i`'s local control state, `M` is the multiset of in-flight signed messages, and `kappa ∈ {Pending, Commit, Abort}` is the commit decision. Let the local projection at zone `i` be `pi_i(g) = (ell_i, M↓_i, kappa)`, where `M↓_i` is the subsequence of messages addressed to `i`. Define `R_cz = { (pi_A(g), g), (pi_B(g), g) | g is a well-typed corridor state }`.

Proof obligation (local-global bisimulation under partial synchrony). Assume partial synchrony: there exists a global stabilization time `GST` and a bound `Delta < ∞` such that after `GST` every authenticated corridor message is delivered within `Delta`; each zone signs at most one verdict per `(operation, phase)`; and timeout transitions delay at least `Delta` for the expected message. Then `R_cz ⊆ ≈_w`.

Proof sketch. A local internal step (timer decrement, non-verdict bundle extension, effect-row growth, or the other zone's internal reduction) is τ on both sides. A local observable `Lock`, signed verdict, `Commit`, `Abort`, or acknowledgement corresponds to a unique global transition under the session type of §10; conversely any global step whose projection is visible at zone `i` is matched locally after at most one send and one receive, both absorbed into a weak step because message buffering is τ and delivery is `Delta`-bounded after `GST`. The single-verdict assumption excludes equivocation, so the local projection cannot branch away from the global decision.

Intuition. A zone's local signed-commitment view is behaviorally the same as the global protocol once the network eventually stops delaying messages without bound.

9.6 Observational Equivalence

Let `T_Op = { Value(v), Paused(E, K, b), Halted(err, mu), SanctionsBlocked(mu), AuthorizationBlocked(mu), OutOfStructuralGas(mu), OutOfCompensationGas(mu), Timeout(mu) }`.

Two closed programs `P` and `Q` are observationally equivalent up to `T_Op`, written `P ≡_obs Q`, when for every closing core context `K`: (i) `K[P]` and `K[Q]` have the same weak traces over `Obs_Op`; and (ii) whenever one reaches a terminal in `T_Op`, the other reaches a terminal with the same constructor and observable payload.

Conjecture (behavioural equivalence). For closed well-typed Op programs, `P ≡_obs Q` if and only if `P ≈_w Q`.

Proof sketch. (⇒) Let R be observational equivalence. Because Op is deterministic, any unmatched observable step would be exposed by a closing context that blocks until exactly that label or terminal constructor, contradicting `P ≡_obs Q`; so R is a weak bisimulation. (⩽) If `P ≈_w Q`, the congruence theorem of §9.4 lifts the relation through every closing core context, and the bisimulation clauses guarantee the same weak traces and the same `T_Op`-terminals.

Intuition. For Op, "no context can tell the programs apart" is exactly the same notion as weak bisimilarity.

9.7 Adequacy Direction (b)

Open proof obligation (direction (b) via weak bisimulation up to τ). Let `r` be an admissible Lex term and let `v` be a verdict. If the Op execution of `[[ r ]]` produces a proof-bundle entry with verdict `v`, then the Lex evaluation of `r` produces the same verdict `v`. The intended proof factors through the verdict projection `pi_V : Obs_Op → {τ} ∪ {(verdict, v)}` that sends every non-verdict observable to τ.

Proof strategy. The relation used in §8.4 should become a weak bisimulation after quotienting labels by `pi_V`. By definition of the silent boundary, structural-gas ticks, non-verdict proof-bundle extensions, and effect-row growth are already τ; the remaining observables must be shown either to preserve the current verdict or to introduce the same verdict on both sides. The reverse implication of §8.4 does not follow until this coinductive argument is closed.

Intuition. Direction (b) works because the proof erases only administrative Op steps; no erased step can manufacture a verdict absent from the Lex source.

---

10. Cross-Zone Execution

A cross-zone operation spans two sovereign kernels. Its correctness has three structural requirements: a linear resource witness that types the lock role, a binary session type whose endpoint projections type each kernel's local view, and a replay specification that reduces verification to pairwise digest comparison. The signed commitment protocol instantiates all three. The present construction is bilateral (Initiator, Responder); generalization to N-ary corridors is future work.

10.1 Indexed Corridor Witnesses

The signed commitment protocol carries the canonical structure of a binary session (Honda, Vasconcelos, Kubo, ESOP 1998): two endpoints, a session protocol type whose endpoint projections type each local participant's view, linear resource handling at each evidence role, and duality between the projections. We adopt the directed-message global-type notation of Honda, Yoshida, and Carbone (POPL 2008) for presentation; the N-ary coherence apparatus of that framework is not used here, since the corridor is bilateral. We write ω for the operation identifier and ε for the corridor epoch.

The global session type `G_corridor(ω, ε)`:

Global and local types. Roles range over `p, q, r`; payload types range over `T` and include `Locked<TensorValue>`. Global types are:

```
G ::= p → q : l(T).G
    | p → q : { l_i(T_i).G_i }_{i in I}
    | G_1 | G_2
    | mu X.G
    | X
    | end
```

The abbreviation `p → q : l(T)` means `p → q : l(T).end`. Local types are:

```
L ::= q!l(T).L
    | q?l(T).L
    | q ⊕ { l_i(T_i).L_i }_{i in I}
    | q & { l_i(T_i).L_i }_{i in I}
    | mu t.L
    | t
    | end
```

Here `q!l(T).L` sends label `l` with payload `T` to `q`, `q?l(T).L` receives it from `q`, ⊕ is internal choice, and & is external choice. A global type is **coherent** when every projection below is defined, every uninvolved branch is mergeable, and every projected send/receive or select/branch action has the matching dual action at its peer.

Projection. Projection `G ↾ r` follows the standard MPST clauses:

```
(p → q : l(T).G) ↾ r =
  q!l(T).(G ↾ p)                  if r = p
  p?l(T).(G ↾ q)                  if r = q
  G ↾ r                           otherwise

(p → q : { l_i(T_i).G_i }_{i in I}) ↾ r =
  q ⊕ { l_i(T_i).(G_i ↾ p) }      if r = p
  p & { l_i(T_i).(G_i ↾ q) }      if r = q
  ⊔_{i in I} (G_i ↾ r)            otherwise
```

Parallel composition and recursion project homomorphically. The merge operator ⊔ is defined only on mergeable local types. The key case is external choice at an uninvolved role:

```
p & { l_i(T_i).L_i }_{i in I} ⊔ p & { l_j(T_j).L'_j }_{j in J}
= p & (
    { l_k(T_k).(L_k ⊔ L'_k) }_{k in I ∩ J}
    ∪ { l_i(T_i).L_i }_{i in I \ J}
    ∪ { l_j(T_j).L'_j }_{j in J \ I}
  )
```

provided shared labels agree on payload types and recursive continuations. For selections, mergeability is stricter: the label sets must coincide branchwise. Any undefined merge renders the global type incoherent. This is the standard side condition preventing a role from being required to distinguish a branch it cannot observe.

Local typing. Endpoint programs are typed by judgements `Γ ; Δ ⊢_p P : L`, where Γ is an unrestricted data environment and Δ is a linear environment. The principal rules are:

```
Γ ⊢ v : T       Γ; Δ ⊢_p P : L
-------------------------------------
Γ; Δ ⊢_p send[q,l](v); P : q!l(T).L

Γ, x:T; Δ ⊢_p P : L
-------------------------------------
Γ; Δ ⊢_p recv[q,l](x); P : q?l(T).L

j in I   Γ ⊢ v : T_j   Γ; Δ ⊢_p P : L_j
-----------------------------------------------------
Γ; Δ ⊢_p select[q,l_j](v); P : q ⊕ { l_i(T_i).L_i }_{i in I}

forall i in I. Γ, x_i:T_i; Δ ⊢_p P_i : L_i
-------------------------------------------------------------------
Γ; Δ ⊢_p offer[q,{ l_i(x_i) ⇒ P_i }_{i in I}] : q & { l_i(T_i).L_i }_{i in I}
```

These are accompanied by the standard guarded recursion and termination rules. When `T = Locked<TensorValue>`, the linear environment Δ is discharged only by the `commit_transfer` and `release_lock` eliminators of §4.10, so the session branch and the resource branch coincide.

The N-ary corridor family. Let the participant set be `{z_0, z_1, ..., z_{N-1}}` with `N ⩾ 2` and `z_0` the initiating coordinator. Write

```
Prep_i =
  z_0 → z_i : TensorRequest(Req).
  z_i → z_0 : Locked_i(Locked<TensorValue>).
  z_0 → z_i : Verdict_0(SignedVerdict).
  z_i → z_0 : Verdict_i(SignedVerdict)
```

and use `Π_{i=a}^b H_i` as meta-notation for right-associated sequencing `H_a.H_{a+1}....H_b`, with the empty product equal to `end`. Define the two uniform decision tails

```
CommitTail^(N) = Π_{i=2}^{N-1} z_0 → z_i : Commit(CommitCert). z_i → z_0 :
 ↪  CommitAck(Ack_i)
AbortTail^(N)  = Π_{i=2}^{N-1} z_0 → z_i : Abort(AbortCert).  z_i → z_0 :
 ↪  AbortAck(Ack_i)
```

and the global family

```
G_corridor^(N) =
```

```
  (Π_{i=1}^{N-1} Prep_i).
  z_0 → z_1 : {
    Commit(CommitCert).
      z_1 → z_0 : CommitAck(Ack_1).
      CommitTail^(N).
      end,
    Abort(AbortCert).
      z_1 → z_0 : AbortAck(Ack_1).
      AbortTail^(N).
      end
  }
```

The former bilateral corridor is not primitive: it is the `N=2` instance of this family, `G_corridor = G_corridor^(2)(Initiator, Responder)`:

```
G_corridor =
  Initiator → Responder : TensorRequest(Req).
  Responder → Initiator : Locked(Locked<TensorValue>).
  Initiator → Responder : Verdict_I(SignedVerdict).
  Responder → Initiator : Verdict_R(SignedVerdict).
  Initiator → Responder : {
    Commit(CommitCert).
      Responder → Initiator : CommitAck(Ack).
      end,
    Abort(AbortCert).
      Responder → Initiator : AbortAck(Ack).
      end
  }

G_corridor(ω, ε) =
  Initiator → Responder : TensorRequest(entity, op, ω, ε).
  Responder → Initiator : Locked<TensorValue, ω, ε>.
  Initiator → Responder : Signed<Verdict_I, ω, ε>.
  Responder → Initiator : Signed<Verdict_R, ω, ε>.
  Initiator → Responder : Commit | Abort.
  end
```

Its projections are:

```
G_corridor(ω, ε) ↾ Initiator =
  !TensorRequest(entity, op, ω, ε).
  ?Locked<TensorValue, ω, ε>.
  !Signed<Verdict_I, ω, ε>.
  ?Signed<Verdict_R, ω, ε>.
  !(Commit | Abort).
  end
```

and at the Responder endpoint (dual):

```
G_corridor(ω, ε) ↾ Responder =
  ?TensorRequest(entity, op, ω, ε).
  !Locked<TensorValue, ω, ε>.
  ?Signed<Verdict_I, ω, ε>.
  !Signed<Verdict_R, ω, ε>.
  ?(Commit | Abort).
  end
```

The linearity discipline of §4.10 types `Locked<TensorValue, ω, ε>` as a linear resource at the Responder endpoint, and the corridor history context of §6.11 admits at most one such lock and at most one local `Signed` witness per side and index. Local receipt of both `Signed` witnesses yields `Verified<ω, ε>`; conflicting same-index signatures yield `Blame<Z, ω, ε>` instead of a second verified state. The eliminator on `Commit` consumes the lock, the eliminator on `Abort` releases it. This is the linear-session-type discipline of Wadler (ICFP 2012) applied to a compliance tensor payload.

Merge is vacuous at N=2. The merge operator `⊔` of the projection clauses above is defined in full generality to handle uninvolved roles in N-ary sessions. At the bilateral corridor there are no uninvolved roles: every global action has exactly one sender and one receiver, and the set of participants is exactly `{p, q}`. The projection algorithm therefore never invokes `⊔`; the otherwise-branch `⊔_{i in I} (G_i ↾ r)` is unreachable at `N=2`. The mechanized projection `bprojection` in `MPSTProjection.v` (`github.com/momentum-sez/op`) reflects this: its constructor set is `{BG_Send, BG_Branch, BG_End}`, with no `BG_Par`, no `BG_Mu`, and no merge invocation site. Projection is total on the binary fragment; the Coq theorem `bprojection_dual` asserts that the two projections are duals of each other, a stronger condition than consistency.

Duality. Let `L` be the grammar of local types above. Define `dual : L → L` by:

```
dual(q!l(T).L)                          = q?l(T).dual(L)
dual(q?l(T).L)                          = q!l(T).dual(L)
dual(q ⊕ { l_i(T_i).L_i }_{i in I})      = q & { l_i(T_i).dual(L_i) }_{i in I}
dual(q & { l_i(T_i).L_i }_{i in I})      = q ⊕ { l_i(T_i).dual(L_i) }_{i in I}
dual(mu t.L)                            = mu t.dual(L)
dual(t)                                  = t
dual(end)                                = end
```

`dual` is a structural operation on the local-type grammar. The N-ary extension is required before session machines support a categorical functor. Its basic properties are mechanized in `SessionDuality.v` (`github.com/momentum-sez/op`): `dual_involution` (`dual ∘ dual = id`), `dual_injective`, `dual_surjective`, and `dual_bijection` (the first three combined).

Theorem 10.1 (Binary duality of projection). Let `G` be a bilateral global session type over participants `{p, q}`, where bilateral means every action is of the form `p → q : …` or `q → p : …`. Then:

1. `bprojection(G, p) = dual(bprojection(G, q))` and symmetrically `bprojection(G, q) = dual(bprojection(G, p))`.
2. Duality at `G` implies coherence at `G` in the sense that every send in one endpoint is matched by a dual receive in the other; no stuck local action exists on a well-formed bilateral global type.

Mechanized as `bilateral_duality` with companion theorems `bilateral_mpst_coherence` and `bilateral_duality_joint` in `MPSTProjection.v` and `SessionDuality.v`. The argument is direct: structural induction on `G` against the four constructor cases of `bgtype`, applying `dual_involution` at the leaves and the projection clauses at the internal nodes.

10.2 Payload Parametricity

The session-safety theorems of §10.3 and §10.4 are proved in Coq over an uninterpreted payload `Parameter payload : Type`; safety is therefore abstract over the payload instance. Instantiating `payload` at the concrete tensor family `Locked<TensorValue, ω, ε> / Signed<V, ω, ε> / Verified<ω, ε>` is an additional structured-payload obligation. The session theorem remains payload-parametric; concrete tensor safety requires the separately stated §7.5 Applicable-fragment meet-monotonicity lemma and the mixed-axis `MeetResult` handling. §8.5 discloses the payload abstraction honestly.

10.3 Binary Corridor Deadlock-Freedom

Theorem 10.2 (Binary corridor deadlock-freedom). Let `G` be a bilateral global session type satisfying the premises of Theorem 10.1, and let `L_p, L_q` be the two projections. Let `Γ_p; Δ_p ⊢_p P_p : L_p` and `Γ_q; Δ_q ⊢_q P_q : L_q` with every carried `Locked<TensorValue>` governed by the linear rules of §4.10. Then the composed two-party configuration `P_p | P_q | Q` under the finite-state session-reduction semantics of `SessionCorridor.v` (`github.com/momentum-sez/op`) does not reach a stuck configuration: every nonterminal reachable state has an enabled communication step.

Mechanized as `deadlock_freedom` in `SessionCorridor.v`. The argument is finite-state reachability: the product automaton of `P_p` and `P_q` under duality has no cycle outside the terminal `(end, end)` state and every non-terminal state has at least one outgoing transition. No fair-delivery or partial-synchrony hypothesis enters Theorem 10.2; that hypothesis enters Open obligation 10.4 below.

10.4 Binary Corridor Session Safety and Commit-Safety

Theorem 10.3 (Binary corridor session safety, payload-abstract). Under the premises of Theorem 10.2, every reachable reduction of `P_p | P_q | Q` preserves session fidelity: each step consumes the next action prescribed by the projections and preserves typability. Every terminal reachable configuration has both endpoints at `end`, empty queues on both sides, and one of Commit or Abort selected on both sides in the abstract session skeleton (no mixed decision, no mixed ack).

Mechanized by the family `session_safety`, `corridor_reaches_terminal`, `no_mixed_decision`, and `no_mixed_ack` in `SessionCorridor.v`. The certificate-core layer is mechanized separately in `BSCInvariants.v` by `finality_certificate`, `abort_certificate`, `terminal_evidence`, `finality_commit_iff_both_accept`, and `recovery_from_terminal_evidence_is_pure`. The session mechanization remains finite-state reachability over an abstract `Parameter payload : Type`; instantiating that payload with `Locked<TensorValue, ω, ε>` and connecting the terminal payload to §7.5 meet-monotonicity is an explicit proof obligation, not part of the payload-abstract theorem.

Open corollary 10.3.1 (Concrete tensor-meet commit safety). Instantiate the payload parameter with `Locked<TensorValue, ω, ε>` / `Signed<V, ω, ε>` / `Verified<ω, ε>`, add the SJN finality certificate and receipt-chain durability premises, and prove that the compliance-tensor outcome at a Commit terminal is the pointwise Applicable-fragment meet of the two endpoints' tensors with `MeetResult` handling on mixed-axis coordinates. Only then does §7.5 meet-monotonicity imply that the committed outcome is no more permissive than either contributor.

Open obligation 10.4 (Corridor commit-safety under partial synchrony). Layer the session-level guarantees of Theorems 10.2-10.3 onto the Sovereign Jurisdiction Network's commit-safety layer: assume partial synchrony in the sense of Dwork, Lynch, and Stockmeyer (1988), meaning there is a global stabilization time GST such that after GST every authenticated corridor message is delivered within a bound $\Delta < \infty$, and assume each zone signs at most one verdict per `(ω, ε)`. Then for any bilateral corridor commit on `(ω, ε)` under these hypotheses:

- SJN invariants (I1)-(I3) hold by typing: `Locked<T, ω, ε>` enforces one active lock per side and index (I1, §6.11), `Signed<V, ω, ε>` enforces one local verdict signature per side and index (I2), `Verified<ω, ε>` is derivable only when both signed witnesses are durable (I3).
- Accountable disagreement should be derivable: any two zones that reach terminal disagreement at the same `(ω, ε)` either lack the required finality certificate, return a malformed-witness obstruction, or expose a conflicting signed artefact. Equivocation by any signer produces `Blame<Z, ω, ε>` only after both conflicting artifacts are imported; before publication it remains a network-layer detection obligation rather than an Op typing fact.

The partial-synchrony hypothesis discharges only the message-delivery liveness aspect; accountable disagreement and the corridor typestate invariants are session-finite and do not use it. Open obligation 10.4's bilateral proof target is written against the §10.7 signed-commitment decomposition; the SJN companion paper names the corresponding finality and accountability obligations at the kernel-network layer.

Subsequent expositions of payload-abstract MPST safety appear in Coppo, Dezani-Ciancaglini, Padovani, and Yoshida (2015), Padovani, Vasconcelos, and Vieira (2014), and Dardha and Gay (2018); Op's contribution here is the specific refinement where the session payload is a compliance tensor valued in a distributive lattice and where §7.5 meet-monotonicity composes the two endpoints' outputs.

10.5 ADGM-DIFC-MAS Worked Example

Take roles `A = ADGM`, `D = DIFC`, and `M = MAS`, with ADGM the coordinator. A concrete three-party corridor is:

```
G_ADGM,DIFC,MAS =
  ADGM → DIFC : TensorRequest(Req).
  DIFC → ADGM : Locked_D(Locked<TensorValue>).
  ADGM → DIFC : Verdict_A(SignedVerdict).
  DIFC → ADGM : Verdict_D(SignedVerdict).
  ADGM → MAS  : TensorRequest(Req).
  MAS  → ADGM : Locked_M(Locked<TensorValue>).
  ADGM → MAS  : Verdict_A(SignedVerdict).
  MAS  → ADGM : Verdict_M(SignedVerdict).
  ADGM → DIFC : {
    Commit(CommitCert).
      DIFC → ADGM : CommitAck(Ack_D).
      ADGM → MAS  : Commit(CommitCert).
      MAS  → ADGM : CommitAck(Ack_M).
      end,
    Abort(AbortCert).
      DIFC → ADGM : AbortAck(Ack_D).
      ADGM → MAS  : Abort(AbortCert).
      MAS  → ADGM : AbortAck(Ack_M).
      end
  }
```

Its projections are:

```
G_ADGM,DIFC,MAS ↾ ADGM =
  !DIFC.TensorRequest(Req).
  ?DIFC.Locked_D(Locked<TensorValue>).
  !DIFC.Verdict_A(SignedVerdict).
  ?DIFC.Verdict_D(SignedVerdict).
  !MAS.TensorRequest(Req).
  ?MAS.Locked_M(Locked<TensorValue>).
  !MAS.Verdict_A(SignedVerdict).
  ?MAS.Verdict_M(SignedVerdict).
  select DIFC {
    Commit(CommitCert).
      ?DIFC.CommitAck(Ack_D).
      !MAS.Commit(CommitCert).
      ?MAS.CommitAck(Ack_M).
      end,
    Abort(AbortCert).
      ?DIFC.AbortAck(Ack_D).
      !MAS.Abort(AbortCert).
      ?MAS.AbortAck(Ack_M).
      end
```

```
  }
```

```
G_ADGM,DIFC,MAS ↾ DIFC =
  ?ADGM.TensorRequest(Req).
  !ADGM.Locked_D(Locked<TensorValue>).
  ?ADGM.Verdict_A(SignedVerdict).
  !ADGM.Verdict_D(SignedVerdict).
  branch ADGM {
    Commit(CommitCert). !ADGM.CommitAck(Ack_D). end,
    Abort(AbortCert). !ADGM.AbortAck(Ack_D). end
  }
```

```
G_ADGM,DIFC,MAS ↾ MAS =
  ?ADGM.TensorRequest(Req).
  !ADGM.Locked_M(Locked<TensorValue>).
  ?ADGM.Verdict_A(SignedVerdict).
  !ADGM.Verdict_M(SignedVerdict).
  branch ADGM {
    Commit(CommitCert). !ADGM.CommitAck(Ack_M). end,
    Abort(AbortCert). !ADGM.AbortAck(Ack_M). end
  }
```

The MAS projection exists only because the Commit and Abort continuations are mergeable at an uninvolved role: both reduce to a branch offered by ADGM with distinct labels and well-typed continuations. A commit trace is:

```
A → D : TensorRequest;
D → A : Locked_D;
A → D : Verdict_A;
D → A : Verdict_D;
A → M : TensorRequest;
M → A : Locked_M;
A → M : Verdict_A;
M → A : Verdict_M;
A → D : Commit;
D → A : CommitAck;
A → M : Commit;
M → A : CommitAck.
```

Every receive in the trace is dual to the next local action in the corresponding projection, all queues are empty at termination, and the same reasoning applies to the abort trace. This is the smallest nontrivial corridor in which mergeability matters: MAS lacks direct observation of ADGM's choice at DIFC, yet its local type remains defined because ADGM forwards the same global branch to MAS.

The old fail-stop side conditions are no longer external assumptions about well-behaved participants. The corridor-indexed typestates of §6.11 make them premises of typing derivations. Skeen (1981) remains the bilateral comparison point, but Op's lock timeout is expressed directly as an event-count eliminator in the

bytecode semantics rather than as coordinator state. The claim here is bounded blocking before finality, not classical non-blocking under arbitrary partitions.

The prior session-type literature (Caires and Pfenning, CONCUR 2010, on the propositions-as-sessions isomorphism; Wadler, ICFP 2012, on linear-logic sessions; Castro-Perez, Seco, and Vasconcelos, POPL 2020, on session-type polymorphism) addresses first-order and higher-order payloads. The contribution at the corridor is the specific refinement where the session-type payload `Locked<T, ω, ε>` carries a compliance tensor valued in a distributive lattice, `Signed<V, ω, ε>` internalizes non-equivocation, and the Commit branch preserves meet-monotonicity (§7.5) by construction.

Zone A (Initiator) Zone B (Responder) TensorRequest(entity, op, ω, ε) Locked<TensorValue, ω, ε> Signed<Verdict_A, ω, ε> Signed<Verdict_B, ω, ε> Commit | Abort Commit_Ack | Abort_Ack Requested LockedReceived Verified<ω, ε> Committed | Aborted Listening Locked<T, ω, ε> Verified<ω, ε> Committed | Released end (dual terminal)

Figure 1. Cross-zone signed commitment as a binary session type. Every arrow is an Op syscall; every typestate transition is a proof-bundle entry. `Locked<T, ω, ε>` is linear at Zone B; bilateral `Signed<V, ω, ε>` witnesses yield `Verified<ω, ε>` plus the finality certificate; eliminators are `commit_transfer` (on Commit) or `release_lock` (on Abort or timeout).

10.6 The Replay Specification

Cross-zone replay is the property that an Op execution can be independently verified by a second zone given only the program, the inputs, and the pack version against which the program was compiled. Verification does not require the second zone to trust the first zone's evaluator; it requires only that the second zone's evaluator conform to the semantics of §6.

Replay is verification. The second zone executes the program against the stated inputs and compares its proof bundle, digest-by-digest, with the first zone's. Agreement accepts the first zone's claim. Disagreement is sufficient evidence of (a) a programmer error in the first zone's claim (the inputs or the program definition was misreported), (b) a semantic bug in one of the two evaluators, or (c) a tampered bundle. In any case, the second zone has a provable answer, not a policy decision.

Open replay theorem. Let `P` be an Op program, `inputs` a typed input record, `pack` a pack digest, and `oracle_log` the content-addressed record of every `external_read` response, `await` callback payload, and deadline resolution consumed along an execution. Let `rho_A` be a proof bundle produced by zone A from `(P, inputs, pack, oracle_log)` and `rho_B` a proof bundle produced by zone B's evaluator from the same four-tuple. The target theorem is `rho_A = rho_B` bit-for-bit, modulo executing-zone identity metadata, under the deterministic semantics, canonical encoding, and gas-threading obligations of §6 and §14. The oracle log is a required replay input; without it, replay is undefined. Deadlines are callback-event-count offsets, not wall-clock offsets; the evaluator has no wall-clock primitive.

10.7 The Signed Commitment Protocol

For operations spanning multiple zones (harbor addition, lawful harbor retirement, bilateral settlement, multi-harbor tensor composition), replay alone is insufficient: each zone must confirm participation before the operation commits. The protocol is a signed commitment discipline with three evidence roles, specified at the Op-programming level:

1. **Lock.** Each zone locks the resources the operation will touch. In Op, a step takes `Linear<T>` values to `Locked<T, ω, ε>` values. The lock records an event-count deadline in the corridor history and is attested with the zone's signature.
2. **Verify.** Each zone independently evaluates the operation's compliance, producing a typed verdict witness `Signed<V, ω, ε>`. Signing requires an empty prior signature history at the same index, so verdicts remain unique per side.
3. **Commit or abort.** The commit phase executes when and only when both verdicts consent and the corridor's finality evidence is present. When both signed verdicts are durably recorded locally, `record` derives `Verified<ω, ε>`. `Locked<T, ω, ε>` values are eliminated by `commit_transfer` on evidence of `Verified<ω, ε>` plus the corridor finality certificate; if either verdict rejects or the deadline expires before finality, `release_lock` eliminates them, returning the underlying `Linear<T>` values. A conflicting same-signer pair derives `Blame<Z, ω, ε>` instead of `Verified<ω, ε>`.

The corridor typestates internalize the three SJN invariants: `Locked<T, ω, ε>` realizes lock monotonicity (I1), `Signed<V, ω, ε>` realizes verdict uniqueness (I2), and `Verified<ω, ε>` realizes verified-entry durability (I3). The type checker enforces these structural properties; finality-certificate formation and cross-zone delivery remain protocol obligations external to local typestate.

The type-level invariant from `Locked<T>` (§4.11) is the protocol's local resource-safety property expressed in the type system: a resource in the `Locked` typestate cannot be discharged other than by producing evidence of the other zone's consent, finality, abort, or timeout. Cross-zone terminal agreement remains the finality-certificate proof obligation, not a fact obtained from local typestate alone.

10.8 Liveness Properties and Their Assumptions

The protocol's structural safety properties now live in the typing rules of §6.11. Liveness still depends on the network and failure model.

Partial synchrony, fail-stop adversary: under partial synchrony in the sense of Dwork, Lynch, and Stockmeyer (1988), there exists a stabilization point after which corridor messages are delivered within a fixed but unknown bound. The Op timeout rule additionally assumes a host-delivered local tick or failure-detector event that advances the event-count deadline. Once both assumptions hold, each side either receives the counterparty's `Signed<V, ω, ε>` witness before `deadline_X(ω, ε)` or fires `E-Lock-Timeout` and releases its `Locked<T, ω, ε>` value. After the finality evidence is present, commit or abort is a pure local

function of the durable certificate. Before that point the protocol is bounded-blocking, not classically non-blocking under arbitrary partitions.

Asynchronous network, fail-stop adversary: termination cannot be guaranteed in finite time without a coordinator or failure-detector oracle (FLP impossibility, Fischer-Lynch-Paterson, 1985). A participant that has sent its verdict and crashed leaves its counterparty unable to decide between commit (if the crashed participant's verdict was `accept`) and abort without additional machinery; no finite event-count deadline can distinguish delay from crash without the eventual-delivery property partial synchrony supplies. Op deployment in asynchronous settings therefore requires either (a) a Paxos-commit coordinator (Gray and Lamport, 2006) or (b) a timeout policy whose false-positive abort rate is tolerated by the application.

Byzantine adversary: the local first deviation handled directly by typing is a second signature attempt inside an honest evaluator: a local attempt to sign a second verdict at the same `(ω, ε)` fails `T-Sign`. Remote equivocation is different. If a zone imports two conflicting signed artifacts under one signer and index, `T-Blame` produces `Blame<Z, ω, ε>` and the corridor cannot derive `Verified<ω, ε>` from that set. The mechanized statement is the local one: the blame precondition is incompatible with invariant I2. Cross-zone safety in §10 is conditioned on each participant signing at most one verdict per (operation, phase); a participant that equivocates, signs two different verdicts to two counterparties, permits two counterparty zones to reach inconsistent commit/abort decisions until the conflicting artifacts are published and imported. Equivocation detection happens at the watcher layer of the companion kernel-network paper via signed-verdict-set publication; it is outside the Op type system. Equivocation resistance is thus a property of the network layer plus watcher attestations, not of the session-type discipline alone; what the watcher and governance layers do with the blame witness after it is produced is the enforcement question.

10.9 Alignment with SJN Commit Obligations

The indexed corridor typestates are the Op translation of the SJN invariants and their commit obligations. SJN's fail-stop obligation derives commit-safety from lock monotonicity (I1), durable verdict storage, and a finality certificate; at the Op layer, `Locked<T, ω, ε>` and its two eliminators express the local resource-safety part as a typing judgment. SJN's partition-heal obligation requires agreement after shared finality and typed obstruction before it; at the Op layer, `Verified<ω, ε>` is a durable local two-signature witness, and `E-Lock-Timeout` supplies the explicit event-count release path when the witness is not obtained before expiry. SJN's accountable-disagreement obligation relies on verdict uniqueness (I2); at the Op layer, `Signed<V, ω, ε>` makes a second local signature ill-typed and `Blame<Z, ω, ε>` reifies any imported conflicting pair. The strengthened Session-Safety theorem is therefore the Op presentation of the bytecode-side pieces of those obligations, not a replacement for the kernel-network finality proof.

10.10 The Sanctions Override

The sanctions-dominance law (§6.12) propagates across replay and the signed commitment protocol: no zone accepts another zone's sanctions verdict via mutual recognition. A receiving zone re-evaluates sanc-

tions independently against its own sanctions lists. The `sanctions_check` effect of a cross-zone receipt is always re-issued by the receiving zone; the receipt is consumed only as evidence of the other zone's separate, non-overridable sanctions evaluation.

---

11. Program Logic for Op

The operational semantics and conservation invariants admit a program logic stated directly over Op states. The point is local reasoning about institutional workflows: a proof should mention only the heap cells, compliance coordinates, and lock witnesses a step touches. We therefore use a Hoare-style logic in the sense refined by local reasoning and separation logic (O'Hearn, Reynolds, and Yang, CSL 2001; Reynolds, LICS 2002), with concurrency rules in the style of Brookes (CONCUR 2004) and rely-guarantee side conditions in the style of Jones (IFIP 1983).

11.1 Compliance-Aware Hoare Triples

Let `Heap` be the finite partial map of heap locations to Op values, and let `K` be the finite partial map of compliance resources: domain verdicts, outstanding obligations, jurisdiction-scoped authorities, and cross-zone lock witnesses. An assertion is a predicate on both:

```
Assert = Heap x K → Prop
```

We write `P, Q ∈ Assert` for compliance-aware assertions. The judgment

```
Γ | κ ⊢ {P} c {Q}
```

reads: under typing environment Γ and ambient compliance context κ, command `c` transforms any well-typed state `(h, κ')` satisfying `P` into a state `(h', κ'')` satisfying `Q`, provided execution terminates without producing a bottom-typed failure token. Assertions may refer simultaneously to heap facts (for example, a treasury row or escrow cell), to compliance facts in κ (for example, that the sanctions or banking coordinate is `Compliant`), and to protocol facts such as signed commit witnesses.

The standard rules lift directly to the compliance-carrying setting. Sequencing composes an intermediate assertion; conditionals split on the guard; primitive calls discharge against their primitive specification. For example:

```
        Γ | κ ⊢ {P} c_1 {R}      Γ | κ ⊢ {R} c_2 {Q}
------------------------------------------------------------ (Seq)
                Γ | κ ⊢ {P} c_1 ; c_2 {Q}
```

The novelty is not Hoare logic itself. The novelty is that the assertions range over the pair "heap plus compliance context," so the postcondition can speak in one sentence about both a write to a state cell and the resulting movement of the 23-domain compliance tensor.

11.2 Separation Conjunction and the Frame Rule

Op's local reasoning principle is separation over both kinds of resource. Write `(h_1, κ_1) # (h_2, κ_2)` when the heap fragments are disjoint and the compliance fragments mention disjoint coordinates, authorities, and lock witnesses. Then `(P * Q)(h, κ)` holds exactly when there exist `(h_1, κ_1)` and `(h_2, κ_2)` such that `h = h_1 ⊎ h_2`, `κ = κ_1 ⊎ κ_2`, `(h_1, κ_1) # (h_2, κ_2)`, `P(h_1, κ_1)`, and `Q(h_2, κ_2)`. The separating conjunction therefore means disjoint operational resources, including disjoint compliance coordinates.

The frame rule is correspondingly two-sorted:

```
        Γ | κ ⊢ {P} c {Q}        disjoint(R, c, κ)
 ------------------------------------------------------ (Frame)
              Γ | κ ⊢ {P * R} c {Q * R}
```

The side condition `disjoint(R, c, κ)` requires that `R`'s support be disjoint from `c`'s read-write footprint in both the heap and the compliance context. Concretely, a filing step that mutates the corporate and licensing coordinates may be framed with a payments assertion; assertions mentioning the same corporate-attestation witness or the same registry row are excluded. This is the local-reasoning law needed for large workflows: a proof about the filing step need not re-prove facts about sanctions, payments, or unrelated locks when those resources are separated.

11.3 Ownership for Locked<T, ω, ε>

For the logic it is convenient to refine the surface `Locked<T>` type with ghost parameters recording the current owner and the protocol evidence carried by the lock. We therefore write `Locked<T, ω, ε>` inside assertions, where ω is the owner principal and ε the current evidence bundle (lock certificate, signed verdicts, or abort witness). The ownership assertion

```
own(x, ω, ε)
```

states exclusive ownership of lock cell `x` at owner ω with evidence ε; exclusivity is the assertion-level form of the linearity invariant.

Ownership admits a transfer rule and two consume rules. Transfer updates the ghost owner when the protocol legitimately hands the lock to another authority:

```
                    handoff(ε, ω, ω')
 ------------------------------------------------------------- (Locked-Transfer)
 Γ | κ ⊢ {own(x, ω, ε) * P} transfer_lock(x, ω', ε) {own(x, ω', ε) * P}
```

Commit consumes the lock on a commit witness and yields a committed resource fact:

```
                    commit_witness(ε, ε_c)
------------------------------------------------------------------------ (Locked-Commit)
Γ | κ ⊢ {own(x, ω, ε) * P} commit_transfer(x, ε_c) {committed(x, ω, ε_c) * P}
```

Abort consumes the lock and restores the underlying linear resource:

```
                    abort_witness(ε, ε_a)
------------------------------------------------------------------- (Locked-Release)
Γ | κ ⊢ {own(x, ω, ε) * P} release_lock(x, ε_a) {linear(x) * P}
```

No rule duplicates `own(x, ω, ε)`, and no rule eliminates it except the two consume rules above. The ownership logic is therefore the proof-theoretic counterpart of the `Locked<T>` typestate from §10.

11.4 Rely-Guarantee for Cross-Zone Commit

Separation alone is insufficient for cross-zone execution because the other zone may step concurrently. We therefore add a rely-guarantee judgment

```
R ⊢ c sat G
```

meaning that if every environment step satisfies the rely relation `R` on combined heap-plus-compliance states, then every step of `c` satisfies the guarantee relation `G`. For the signed commitment protocol of §10.7, the rely condition `R_corr` says that counterparties may only append well-signed verdicts, may sign at most one verdict per `(operation, role)`, and may refine the foreign compliance tensor only by the meet-monotone corridor rule. The guarantee `G_corr` says that the local endpoint either preserves `own(x, ω, ε)`, transfers it once under an authenticated handoff, consumes it exactly once into `committed(x, ω, ε_c)`, or consumes it exactly once into `linear(x)` on abort.

Open theorem schema (commit-protocol soundness). Let `c_BSC` be the Op program implementing the corridor protocol of §10.7. Assume both endpoints satisfy `R_corr ⊢ c_BSC sat G_corr`, begin from separated ownership assertions `own(x_A, ω_A, ε_A) * own(x_B, ω_B, ε_B)`, and exchange only authenticated messages whose finality certificate is durable and replayable. Then every terminal execution satisfies exactly one of the following:

1. Both endpoints hold committed postconditions, and no `Locked` resource remains.
2. Both endpoints hold released linear-resource postconditions, and no `Locked` resource remains.

In particular, no execution reaches a mixed state in which one endpoint has consumed its lock by commit while the other has restored the underlying linear resource by abort.

Proof sketch. The proof is by phase induction on lock, verify, and commit/abort. The separation rules isolate the local lock ownership facts; the rely relation excludes foreign interference that would duplicate

a lock witness, rewrite a signed verdict, or strengthen the foreign tensor beyond meet-monotonicity; the guarantee relation restricts the local endpoint to the three ownership moves above. Session safety from §10.4 removes unmatched protocol branches, and the `Locked<T>` consume rules remove the only two terminal possibilities. Hence any terminal state is either commit/commit or abort/abort, never commit/abort.

11.5 Verification-Condition Generation

The proof system admits a standard compositional VC generator. Write `VC(Γ | κ ⊢ {P} c {Q})` for the verification condition of a triple. The generator is syntax-directed:

```
VC({P} skip {Q})                                ≡  P => Q
VC({P} c_1 ; c_2 {Q})                           ≡  exists R. VC({P} c_1 {R}) /\ VC({R} c_2 {Q})
VC({P} if b then c_t else c_f {Q})              ≡  VC({P /\ b} c_t {Q}) /\ VC({P /\ not b} c_f {
VC({P} prim {Q})                                ≡  Spec_prim(P, Q)
```

Separating conjunction contributes disjointness side conditions; cross-zone steps contribute rely-guarantee obligations; primitive contracts contribute first-order implications over finite maps, digest equalities, signature predicates, and the 23-domain tensor coordinates. These conditions are compositional and naturally SMT-dischargeable: heap footprints compile to finite-map constraints, compliance coordinates compile to a finite product of uninterpreted-sort predicates, and disjointness compiles to non-aliasing plus coordinate-inequality obligations. Routine Hoare obligations compile into a form the host can discharge automatically; the primitive specifications and corridor laws remain the trusted inputs.

11.6 Iris Remark

The resource interpretation above is concrete and first-order, but its shape is the one Iris abstracts: a separation logic whose assertions are predicates over a resource algebra with ownership, framing, and ghost state. In the sense of Krebbers et al. (*Iris*, JFP 2018), Op's logic is therefore a concrete instance of the Iris pattern, with one resource algebra for heap fragments, one for compliance-context fragments, and authoritative ghost elements for `Locked` ownership and signed commit evidence.

---

12. Worked Examples

The examples use primitive names from the prelude; §4 gives the definitions and types.

12.1 Entity Incorporation (Single-Jurisdiction, No Suspension)

Entity formation in a jurisdiction that does not require a pre-flight consent step. The program has four steps: entity creation, treasury creation, bank-account creation, and status activation.

```
op entity.incorporate for sc
```

```
inputs { legal_name: String, entity_type: String, treasury_currency: String }
outputs { entity_id: EntityId, treasury_id: String }
effects { sanctions_check; sovereign_write; }
contracts { requires domains [corporate, sanctions]; ensures domains [corporate]; }
do {
  step entity_create
    : { legal_name: String, entity_type: String, jurisdiction: JurisdictionId }
    -> { id: EntityId }
    ! { sovereign_write }
    create.entity({ name: legal_name, entity_type: entity_type, jurisdiction: "sc" })
    compensate { update.entity_status({ entity_id: entity_create.id, status: "DELETED" }); };
  run treasury = create.treasury({
    entity_id: entity_create.id,
    currency: treasury_currency
  });
  run account = create.bank_account({
    entity_id: entity_create.id,
    treasury_id: treasury.id,
    account_type: "checking",
    currency: treasury_currency
  });
  run activate = update.entity_status({
    entity_id: entity_create.id,
    status: "ACTIVE"
  });
  return { entity_id: entity_create.id, treasury_id: treasury.id };
}
```

The compiled signature is checked against the corporate and sanctions domains of the `sc` jurisdiction's pack. The `entity_create` step may start before sanctions screening (§6.12 exception) because the entity does not yet exist; post-flight sanctions evaluation confirms the entity's compliance before the status transitions to `ACTIVE`.

12.2 Share Issuance (Guarded Consent, Compensation)

A capital-increase operation with conditional governance: a required member resolution triggers a consent step whose outcome is awaited; otherwise the step is guarded away.

```
op capital.increase for _default
inputs {
```

```
  entity_id: EntityId,
  increase_amount: MoneyAmount,
  new_shares_count: Int,
  par_value_per_share: MoneyAmount,
  issue_price_per_share: MoneyAmount,
  share_class: String?,
  member_resolution_required: Bool?
}
effects { sanctions_check; sovereign_write; governance_request; }
contracts {
  requires domains [corporate, sanctions];
  ensures domains [corporate, securities];
}
do {
  let share_class_name: String = coalesce(share_class, "ordinary");
  let needs_member_vote: Bool = coalesce(member_resolution_required, false);
  run board_resolution = consent.board_resolution({
    entity_id: entity_id,
    operation_type: "CAPITAL_INCREASE_BOARD_RESOLUTION"
  });
  choose {
    when needs_member_vote == true -> {
      run member_resolution = consent.member_resolution({
        entity_id: entity_id,
        operation_type: "CAPITAL_INCREASE_MEMBER_RESOLUTION"
      });
    }
    else -> {
      let member_resolution: { consent_id: String? } = { consent_id: null };
    }
  }
  run registrar_filing = filing.registry_amendment({
    entity_id: entity_id,
    new_share_capital: increase_amount,
    new_shares_count: new_shares_count
  })
  compensate {
    filing.registry_reversal({
      filing_id: registrar_filing.filing_id,
```

```
      reason: "ROLLBACK"
    });
  };
  run share_register_update = update.cap_table({
    entity_id: entity_id,
    new_shares_count: new_shares_count,
    par_value: par_value_per_share,
    issue_price: issue_price_per_share,
    share_class: share_class_name,
    registry_reference: registrar_filing.filing_id
  });
  return { filing_id: registrar_filing.filing_id };
}
```

The registrar filing carries a compensation that reverses it if the cap-table update fails or the program aborts. The filing commits to the proof bundle before the cap-table update begins; if the cap-table fails, compensation runs in reverse topological order (cap-table first, a no-op if not committed; filing reversal second).

12.3 Cross-Zone Letter of Credit (Participants, Suspension, Cross-Zone Three-Phase Commit)

A letter-of-credit operation across two zones, with explicit participants, bilateral approval, and signed commitment of a fiscal transfer. The operation suspends on two events: buyer sanctions verification, and LC issuance confirmation.

```
op trade.letter_of_credit for singapore
inputs {
  seller_id: EntityId,
  buyer_id: EntityId,
  issuing_bank: String,
  advising_bank: String,
  goods_description: String,
  total_amount: MoneyAmount,
  currency: String,
  expiry_date: Date
}
participants {
  seller: SourceZone(seller_id);
  buyer:  DestinationZone(buyer_id);
}
approval bilateral;
```

```
effects { sanctions_check; fiscal_transfer; sovereign_write; await verification.completed; aw
contracts {
  requires domains [trade, sanctions, banking];
  ensures domains [trade, banking, tax];
}
do {
  in seller_zone {
    run seller_verification = check.entity_registration({
      entity_id: seller_id
    });
  }
  in buyer_zone {
    run buyer_sanctions : Await<verification.completed, ScreeningVerdict> =
      screening.sanctions({ subject_id: buyer_id });
  }

  run invoice_generation = trade.invoice_create({
    seller: seller_id,
    buyer: buyer_id,
    total: { currency: currency, value: total_amount },
    goods_description: goods_description
  });

  run lc_issuance : Await<document.signed, LcCertificate> =
    trade.lc_issue({
      applicant: buyer_id,
      beneficiary: seller_id,
      amount: { currency: currency, value: total_amount },
      issuing_bank: issuing_bank,
      advising_bank: advising_bank,
      expiry_date: expiry_date
    })
    compensate {
      trade.lc_revoke({
        lc_id: lc_issuance.lc_id,
        reason: "OP_ABORTED"
      });
    };
```

```
  par {
    invoice_document = document.commercial_invoice({
      reference_id: invoice_generation.invoice_id
    });
    duty_assessment = tax.customs_duty_assessment({
      entity_id: seller_id,
      gross_amount: total_amount,
      currency: currency
    });
  }

  return {
    invoice_id: invoice_generation.invoice_id,
    lc_id: lc_issuance.lc_id
  };
}
```

The program declares two participants, each with a role mapping to a zone in the signed commitment protocol. The `bilateral` approval mode commits the LC only when both zones sign their verdicts and finality evidence is present. `await verification.completed` on the sanctions check suspends the buyer-zone portion until the sanctions authority returns a verdict; `await document.signed` on LC issuance suspends until the issuing bank confirms. Both suspensions are typed: the compiler checks that the payload types (`ScreeningVerdict`, `LcCertificate`) match the callback registry and that the rest of the program handles each payload's possible verdicts. The compensation on LC issuance pre-commits a revocation: if the program aborts after LC issuance but before final commit, the issuance reverses.

12.4 Multi-Harbor Composition (Participants Across Three Zones)

A bilateral settlement between two multi-harbor entities (seller in jurisdictions A and B, buyer in jurisdictions C and D) executed through a corridor between A and C. The program composes the seller's tensor (meet of A and B) with the buyer's tensor (meet of C and D) and takes the corridor-translated meet; the resulting verdict governs whether the settlement proceeds.

```
op corridor.settle for _default
inputs {
  seller_id: EntityId,
  buyer_id: EntityId,
  corridor: String,
  amount: MoneyAmount,
  currency: String
```

```
}
participants {
  seller: Participant(seller_id);
  buyer:  Participant(buyer_id);
}
approval unanimous;
effects { sanctions_check; fiscal_transfer; }
contracts {
  requires domains [corporate, sanctions, banking, payments, clearing, settlement];
  ensures domains [banking, settlement];
}
do {
  in seller_home_zone {
    run seller_tensor = tensor.compose_multi_harbor({
      entity_id: seller_id
    });
  }
  in buyer_home_zone {
    run buyer_tensor = tensor.compose_multi_harbor({
      entity_id: buyer_id
    });
  }
  run settlement_tensor = tensor.meet_across_corridor({
    corridor: corridor,
    lhs: seller_tensor,
    rhs: buyer_tensor
  });
  choose {
    when settlement_tensor.status == "Compliant" -> {
      run transfer = settlement.execute({
        sender: buyer_id,
        receiver: seller_id,
        amount: { currency: currency, value: amount }
      })
      compensate {
        settlement.reverse({
          transfer_id: transfer.transfer_id,
          reason: "SETTLEMENT_REVERSED"
        });
```

```
      };
      return { transfer_id: transfer.transfer_id, status: "SETTLED" };
    }
    else -> {
      return { transfer_id: "", status: "BLOCKED" };
    }
  }
}
```

The program illustrates meet-monotonicity (§7.5): the settlement tensor is the corridor-translated meet of the two composed tensors; no domain's settlement verdict is more permissive than either contributing home tensor's. If any contributing harbor's sanctions verdict is `NonCompliant`, the settlement's sanctions verdict is `NonCompliant`, and the guard `settlement_tensor.status == "Compliant"` evaluates to `false`.

12.5 Mutual Recognition Across Heterogeneous Domain Sets (Corridor Translation phi)

An entity incorporated in zone A (ADGM) seeks parallel registration in zone B (Seychelles IBC) under a bilateral corridor whose translation `phi_{A,B} : D_A → D_B` is partial. The two zones carry overlapping but non-identical domain sets: `D_A = { corporate, aml, beneficial_ownership, sharia_compliance, sanctions }` and `D_B = { corporate, aml, beneficial_ownership, tax_residency, sanctions }`. The corridor declares `phi(corporate) = corporate`, `phi(aml) = aml`, `phi(beneficial_ownership) = beneficial_ownership`, `phi(sanctions) = sanctions`, and leaves `phi(sharia_compliance)` undefined. The domain `tax_residency` is in `D_B` but has no preimage under `phi`; zone B evaluates it independently from its own pack. The composed tensor `T_AB(d_B)` has four cases: (i) `d_B` in the image of `phi` yields `T_A(phi^{-1}(d_B)) meet T_B(d_B)`, the pointwise meet across the two zones on Applicable cells; (ii) `d_B` in `D_B` but outside the image of `phi` yields `T_B(d_B)` alone, as zone B evaluates without a zone A counterpart; (iii) `sharia_compliance` has no image in `D_B` and surfaces as a structured `MissingTranslation(lhs_only)` record, carrying the witness that `phi` does not translate it and that it is not used for authorization in this corridor; (iv) any domain outside `D_A union D_B` is undefined for both zones and elided from the composed tensor entirely.

```
op mutual_recognition.register for adgm
inputs {
  entity_id: EntityId,
  target_zone: String,
  corridor: String
}
participants {
```

```
  adgm_entity:       SourceZone(entity_id);
  seychelles_branch: DestinationZone(entity_id);
}
approval bilateral;
effects { sanctions_check; sovereign_write; proof_emit; }
contracts {
  requires domains [corporate, aml, beneficial_ownership, sanctions];
  ensures  domains [corporate, aml, beneficial_ownership, sanctions];
}
do {
  in adgm_zone {
    run t_a = tensor.fetch({
      entity_id: entity_id,
      zone: ”adgm”
    });
  }
  in seychelles_zone {
    run t_b = tensor.fetch({
      entity_id: entity_id,
      zone: ”seychelles”
    });
  }
  run phi = corridor.fetch_translation({
    corridor: corridor,
    source_domains: ”adgm”,
    target_domains: ”seychelles”
  });
  run t_ab = tensor.compose_via_phi({
    phi: phi,
    lhs: t_a,
    rhs: t_b,
    untranslated_policy: ”obstruct_if_required”
  });
  choose {
    when t_ab.image_verdict == ”Compliant” -> {
      run registration = filing.foreign_branch_register({
        entity_id: entity_id,
        target_zone: target_zone,
        composed_tensor: t_ab
```

```
      });
      return { registration_id: registration.filing_id, status: "REGISTERED" };
    }
    else -> {
      return { registration_id: "", status: "BLOCKED" };
    }
  }
}
```

The corridor translation `phi` is fetched as program input; it is never inferred. A composition lacking a declared `phi` is rejected at the `compose_via_phi` boundary (§7.5). The `untranslated_policy: "obstruct_if_required"` argument makes the ADGM-only domain `sharia_compliance` an explicit non-authorization record: zone B has no pack rule to evaluate against, so the composed tensor carries a `MissingTranslation(lhs_only)` result at that coordinate along with the witness that `phi` does not translate the domain. `tax_residency`, dual, passes through from zone B alone. The guard `t_ab.image_verdict == "Compliant"` ranges only over the declared required domains and B-only entries that zone B evaluates; any missing translation on a required domain blocks ordinary execution.

The composed tensor in the Compliant scenario is:

domain	source	T_A	T_B	T_AB
corporate	both	Compliant	Compliant	Compliant
aml	both	Compliant	Compliant	Compliant
beneficial_ownership	both	Compliant	Compliant	Compliant
sanctions	both	Compliant	Compliant	Compliant
sharia_compliance	A only	Compliant	undefined	MissingTranslation
tax_residency	B only	undefined	Compliant	Compliant

Under meet-monotonicity the composed tensor at any `d_B` in the image of `phi` is no more permissive than either contributing entry: `T_AB(corporate) = Compliant meet Compliant = Compliant`, and if instead `T_B(corporate) = NonCompliant` the composed entry collapses to `NonCompliant`, regardless of the ADGM verdict. The `MissingTranslation` record on `sharia_compliance` is witness-bearing, not verdict-bearing: the proof bundle records the triple `(domain: sharia_compliance, phi_coverage: false, witness: corridor.phi.digest)`, so a third-party verifier re-running the corridor protocol confirms the claim that zone B was not asked, and could not have been asked, to rule on sharia compliance under this corridor. The proof-bundle extract for step `t_ab` is:

```
proof_bundle {
  step: "t_ab",
  primitive: "tensor.compose_via_phi",
```

```
  phi_digest: "phi.adgm-seychelles.2026-04.blake3:7f2a...",
  translated_domains: ["corporate", "aml", "beneficial_ownership", "sanctions"],
  lhs_only_domains:   [{ domain: "sharia_compliance", meet_result: "MissingTranslation",
                         reason: "phi_not_defined", used_for_authorization: false }],
  rhs_only_domains:   [{ domain: "tax_residency", verdict: "Compliant",
                         reason: "evaluated_rhs_pack_only" }],
  image_verdict:      "Compliant",
  signature:          "host-sig:..."
}
```

The example demonstrates three capabilities absent from the flat, defeasible, and sanctions examples above: a partial corridor translation as first-class input, a machine-checkable missing-translation record when `phi` does not cover a domain, and a composed verdict ranging over heterogeneous domain sets without either side silently defaulting the other's claims.

12.6 Case Study: ADGM⊠DIFC Commit

This section operationalizes the Lex case study of the same title. The source rule layer is fixed there: ADGM screens against OFAC plus the ADGM list, DIFC screens against the UN consolidated list plus the DIFC list, ADGM fills a "fit and proper" hole, DIFC fills an "approved person" hole, and both sides derive a reporting deadline from the committed event. The present concern is the compiled bytecode and the corridor protocol.

A representative compiled sequence, rendered in a readable pseudocode form for exposition (the canonical wire format of §3.7 is CBOR-tagged AST; the listing below names the operations in evaluation order rather than showing their CBOR bytes):

```
0:  LOAD_TX tx_id
1:  LOAD_PARTY adgm_manager
2:  SANCTIONS.CHECK OFAC
3:  SANCTIONS.CHECK ADGM_LIST
4:  ASSERT_CLEAR
5:  LOAD_PARTY difc_counterparty
6:  SANCTIONS.CHECK UN_LIST
7:  SANCTIONS.CHECK DIFC_LIST
8:  ASSERT_CLEAR
9:  PCAUTH.LOAD fit_and_proper_adgm
10: PCAUTH.VERIFY ADGM.FSRA
11: PCAUTH.LOAD approved_person_difc
12: PCAUTH.VERIFY DIFC.DFSA
13: PCAUTH.LOAD MutualRecognition
```

```
14: BRIDGE.VERIFY ADGM DIFC CommitReady(tx_id)
15: DEADLINE.DERIVE report_deadline tx_id
16: REPORT.ASSERT report_deadline
17: TENSOR.MEET
18: LOCK tx_id
19: SIGN.VERDICT ADGM
20: SEND MAS
21: SIGN.VERDICT DIFC
22: SEND MAS
23: MAS.DECIDE
24: COMMIT | ABORT
```

Instruction 18 introduces the linear witness `Locked<TensorValue, ω=tx_id, ε=0>`. Instructions 19 and 21 emit `Signed<Verdict, ω, ε>` records bound to the same transaction identifier ω and phase counter ε. The lock is therefore consumed only by the terminal commit branch and restored only by the abort branch.

PCAuth material travels through the corridor as proof-bundle attachments rather than as ambient trust. The ADGM fill for `fit_and_proper_adgm`, the DIFC fill for `approved_person_difc`, and the bridge witness `MutualRecognition` are transported beside the signed verdict, content-addressed by the same ω. A receiving kernel replays the witness verification before accepting the foreign verdict; Op ships the evidence, not a privileged interpretation of it.

The following global session is an illustrative n-party target for the three parties ADGM, DIFC, and MAS:

```
G_commit =
  ADGM -> DIFC : TensorRequest(tx_id).
  DIFC -> ADGM : Locked<TensorValue, omega=tx_id, epsilon=0>.
  ADGM -> MAS  : Signed<Verdict, omega, epsilon>.
  DIFC -> MAS  : Signed<Verdict, omega, epsilon>.
  MAS  -> ADGM : Commit | Abort.
  MAS  -> DIFC : Commit | Abort.
  end
```

with projections

```
G_commit | ADGM =
  !TensorRequest(tx_id).
  ?Locked<TensorValue, omega=tx_id, epsilon=0>.
  !Signed<Verdict, omega, epsilon>.
  ?(Commit | Abort).
  end
```

```
G_commit | DIFC =
  ?TensorRequest(tx_id).
  !Locked<TensorValue, omega=tx_id, epsilon=0>.
  !Signed<Verdict, omega, epsilon>.
  ?(Commit | Abort).
  end

G_commit | MAS =
  ?Signed<Verdict, omega, epsilon>.
  ?Signed<Verdict, omega, epsilon>.
  !(Commit | Abort).
  !(Commit | Abort).
  end
```

This is not part of the Qed-closed bilateral calculus of §10. It sketches the intended N-ary extension named as open work in §14.21. MAS does not reinterpret the local law; in the target design it verifies signatures, witness digests, and phase coherence, then broadcasts the decision both zones are typed to consume.

The proof obligations again split by layer. Mechanized today: the sanctions and filled-hole compilation cases and the binary, payload-parametric session skeleton. The linear treatment of the lock is mechanized over the abstract BSC history context, not over this concrete three-party payload. SMT-discharge targets: deadline arithmetic, digest equality, and the finite guard conditions at `BRIDGE.VERIFY` and `REPORT.ASSERT`. PCAuth-filled: the ADGM "fit and proper" judgment, the DIFC "approved person" judgment, and the `MutualRecognition` bridge witness imported from the Lex layer. Under partial synchrony plus host-delivered deadline events, EUF-CMA security of Ed25519 for all `Signed` and PCAuth objects, correctness of the sanctions and deadline oracles, and durable finality-certificate dissemination, the target accepting branch is sound and accountable. Soundness here is a design obligation: no commit should occur unless replay of the compiled rule layer reproduces the same passing tensor and both zone signatures bind to the same (ω, ε). Accountability means every non-mechanical input remains attached to a signer, scope, digest, and timestamp in the proof bundle.

12.7 Non-Example: Route-Coherence Failure Across Bilateral Agreements

Consider an entity harbored in `J_1`, `J_2`, `J_3` with three bilateral corridors:

- `C_{12}`: `(R_{12} = {Sanctions, AML}, phi_{12})`: `J_1 → J_2`
- `C_{23}`: `(R_{23} = {Sanctions, Securities}, phi_{23})`: `J_2 → J_3`
- `C_{13}`: `(R_{13} = {Sanctions, AML, Securities}, phi_{13})` , `J_1 → J_3`, negotiated bilaterally

J_2's domain Banking maps to J_3's FinancialServices under phi_{23}; J_1's Banking maps to J_3's Banking directly under phi_{13}. Route an entity passport A → C two ways: directly across C_{13}, or through B (C_{12} then C_{23}).

```
let p_13 = corridor_cross(passport_1, C_13)
let p_12 = corridor_cross(passport_1, C_12)
let p_23 = corridor_cross(p_12, C_23)
assert_eq(domain_image(p_13), domain_image(p_23))
```

Expected diagnostic:

```
DescentObstruction {
  kind: RouteCoherenceFailure,
  domain: Banking,
  direct_image:   J_3.Banking,
  composed_image: J_3.FinancialServices,
  witnesses: [phi_{13}(Banking), phi_{23}(phi_{12}(Banking))],
  resolution: renegotiate one of {C_12, C_23, C_13}
}
```

The pipeline returns the typed obstruction rather than silently picking a winner. The descent obligation on the EntityTrajectoryGraph is a first-class kernel value enforced by the semantics.

---

13. Prior Art

The design space for Op is crowded. The six sections below place Op relative to its most relevant neighbors.

13.1 The Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM), specified in the Ethereum Yellow Paper (Wood, 2014) and mechanized in formal frameworks (Hildenbrandt et al., 2018), is the most cited precedent for typed bytecode with a gas model. EVM is a stack-based machine with a fixed opcode set, per-opcode gas cost, atomic all-or-nothing transactions, and a deterministic reduction relation. It supports cross-participant verification (every Ethereum node independently executes every transaction), and its gas mechanism is extensively studied.

Op shares with EVM deterministic reduction and a metered, ex-ante-bounded gas regime. Op's split of gas into a structural axis (bounded at compile time by the AST) and an extensional axis (parameterized by runtime cardinalities via certificates) is sharper than EVM's per-opcode schedule, which conflates them: EVM's gas costs are per-instruction fixed charges plus storage/memory expansion surcharges, all metered at

runtime. Op differs from EVM in four respects. First, Op is a compliance-workflow substrate, not a general-purpose computation substrate; its primitive set reflects the domain (consent, screening, document generation, filing) rather than the general-computation set (arithmetic, memory, storage primitives). Second, Op has typed effects (§4.9) that EVM does not; EVM's effect discipline is implicit in its opcode partitioning (state-reading versus state-writing) rather than in its type system. Third, Op has explicit suspension semantics (§6.8) for callback events; EVM has no suspension, as every transaction is atomic and either completes or fails within a single block. Fourth, Op's cross-zone replay model is pairwise, not global; there is no global operation ordering. EVM's replay is the replay of a total global order across all participants.

The 2016 DoS attack on Ethereum's `EXTCODESIZE`, `BALANCE`, and `SLOAD` opcodes (addressed by the Tangerine Whistle and Spurious Dragon hard forks, both in 2016) is the canonical historical demonstration that an EVM-style flat per-opcode schedule can be exploited when primitive cost models under-price the actual resource consumption. EIP-150 (Buterin, 2016) raised the per-call cost of the affected opcodes by a factor of forty after the attacker spent $1,000 in transaction fees to consume tens of thousands of dollars of validator compute. Op's structural-vs-extensional split is an explicit response to this lesson: the structural gas is provably bounded by the AST, so it cannot be under-priced by a static schedule; the extensional gas is parameterized by a cardinality certificate supplied at dispatch, so its price is a function of runtime input size rather than a fixed opcode surcharge. The open-problem subsection on gas pricing under adversarial cardinality states the remaining residual: an attacker supplying a cardinality certificate under-reporting the true runtime size still forces a mid-execution abort with compensation. The safety response is correct; the pricing response (disincentivize adversarial under-reporting via posted-bond cardinality schemes or optimistic-rollup-style challenge periods) remains open.

13.2 WebAssembly

WebAssembly (Haas et al., 2017) is a typed, low-level bytecode designed for portability and determinism across host environments. It has a well-specified operational semantics, a two-level type system, and structured control flow. Several compliance-framework systems have adopted WebAssembly as their execution target.

Op shares with WebAssembly the typed-bytecode premise and determinism emphasis. Op differs in two structural respects. First, WebAssembly is general-purpose (Turing-complete by intent, recursion primitive); Op is domain-specific and deliberately bounded at its recursion forms. Second, WebAssembly has no notion of compensation, suspension on external events, or jurisdiction-scoped primitives; these are library or host concerns, left to embedders to standardize. Op lifts them into primitive grammar, on the view that leaving them to embedders produces the same fragmentation YAML workflow formats produce.

A natural question: should Op be implemented as a WebAssembly module? The correct answer is: only as a backend target in the sense of §15. The compliance semantics (suspension, compensation locality, sanctions dominance, participant composition) are not delegated to WebAssembly primitives. The compiler

preserves those Op-defined semantics while lowering control flow and primitive dispatch into a portable WebAssembly artifact.

13.3 Michelson

Michelson (Allombert et al., 2019) is the bytecode of the Tezos blockchain: a stack-based, strongly typed language designed for formal verification, with a type system simple enough for mechanical checking and operational semantics specified in Coq. Michelson has no compensation construct; its failure story is `FAILWITH`, which aborts the current transaction and reverts state. Its execution is transaction-atomic, as EVM's is.

Op sits closer to Michelson than to EVM: both prioritize typed bytecode with formal semantics, both treat the type system as a load-bearing safety layer, and both reject the ambient-effect discipline typical of general-purpose bytecodes. The Op design choices Michelson does not make (non-atomic suspension, compensation attached to step bodies, jurisdiction-scoped primitives) are not Michelson oversights; they are out of scope for a blockchain bytecode. Op is a compliance-workflow bytecode; the scope difference is the design difference.

13.4 Solidity Intermediate Representation

Yul is the intermediate representation of the Solidity compiler: a typed, lightly structured language targeting EVM and eWASM backends, designed for optimization and verification. Its semantics are specified operationally over a small instruction set with explicit function calls, switches, and loops.

Op differs from Solidity IR as it differs from EVM: the domain primitives, the explicit suspension, the compensation locality, and the meet-monotonic composition are not expressible in Solidity IR without library-level extensions that would recapitulate the language-design problem. Op programs are also authored, not generated; Solidity IR is typically a target for a front-end compiler (Solidity, Vyper), not an authoring surface.

13.5 Extracted Coq Interpreters

Verified-compiler work provides the template for languages whose operational semantics are mechanized and whose implementations are extracted with end-to-end preservation proofs. CompCert (Leroy 2006, 2009) mechanizes back-end preservation for a C subset. CompCertTSO (Sevcik, Vafeiadis, Zappa Nardelli, Jagannathan, Sewell, 2013) extends preservation into a relaxed-memory concurrent setting, establishing that proof-producing compilation survives non-sequential execution semantics, the closest prior analogue to Op's cross-zone replay obligation, where each zone's evaluator is an independent observer of the same program trace. Proof-carrying code (Necula, POPL 1997) supplies the foundational discipline for shipping a program together with a machine-checkable proof of its safety properties; Op's proof-bundle discipline generalizes PCC, extending it from a single safety predicate to a structured, content-addressed record of verdict-introducing steps and effect-typed side effects. CakeML (Kumar, Myreen, Norrish, Owens, POPL

2014) extends verified-compilation to a functional source language with a verified chain from HOL semantics through compilation to x86 machine code. The Bedrock line and various Isabelle/HOL small-language formalizations cover adjacent ground.

The closest Op analogue is an extracted Coq interpreter for the admissible fragment. The Op operational semantics (§6) are stated in the style appropriate for such a mechanization; the conservation invariants (§7) stand as theorems whose proofs could be discharged in a proof assistant. The natural next step beyond this paper: mechanize the semantics, mechanize the invariants, and extract a reference interpreter. The paper does not claim the mechanization is present; it claims the semantics are stated in a form that admits it.

F* (Swamy et al., POPL 2016) and its smart-contract specialization (Bhargavan et al., PLAS 2016) mechanize smart-contract semantics in a general-purpose effectful dependent type theory, with extraction to OCaml or to EVM bytecode. Op differs in two respects. The effect vocabulary is a fixed finite set rather than an open-ended algebra; this is deliberate, because admissible compliance effects are bounded by the primitive taxonomy, and the sanctions-dominance law requires a distinguished bottom-propagating effect no handler may recover from. The target is a compliance-workflow bytecode with typed suspension and scoped compensation rather than a general-purpose smart-contract runtime; the primitives reflect an institutional vocabulary (consent, screening, document generation, filing) rather than a general-computation vocabulary. Verification at the host level (the operating system hosting the Op evaluator) is outside this paper's scope. seL4 (Klein et al., SOSP 2009) demonstrates that microkernels admit full functional verification; the Op evaluator is a user-space component whose correctness presupposes a trusted host. The companion kernel-network paper addresses the host-trust story.

13.5.1 JVM bytecode verifier and eBPF static analysis

The Java Virtual Machine's bytecode verifier (Lindholm, Yellin, Bracha, Buckley, *Java Virtual Machine Specification* SE 8) is the canonical example of a typed-bytecode verifier deployed at scale: every classfile loaded into a JVM is statically checked for type safety, stack-depth consistency, control-flow well-formedness, and linkage integrity before any instruction executes. Stata and Abadi (*A Type System for Java Bytecode Subroutines*, POPL 1998) give the canonical type-theoretic account of the JVM's subroutine-calling discipline; Freund and Mitchell (*A Type System for Java Bytecode with Dynamic Loading*, TOPLAS 2003) extend this to a formal soundness theorem against the JVM's dynamic loader. The Kildall-style dataflow fixed-point algorithm used by the JVM verifier (Kildall 1973; Palsberg-Schwartzbach) is the prototype for all later bytecode-verification work.

eBPF (extended Berkeley Packet Filter; Starovoitov, Linux kernel 3.18, 2014) runs untrusted user-provided bytecode inside the Linux kernel after passing an in-kernel verifier that proves termination, bounded stack/heap, bounded iteration, and absence of ambiguous memory access. Gershuni, Amit, Sagi, et al. (*Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions*, PLDI 2019) give a polyhedral domain for the verifier; Nelson, Bornholt, Gu, et al. (*Specification and Verification in the Field: Applying Formal Methods to BPF Just-in-Time Compilers in the Linux Kernel*, OSDI 2020) proves preservation across the JIT.

Op's verifier story is closest to JVM-and-eBPF in technology stack, and farther from CompCert in what it proves. Like the JVM verifier, an Op verifier takes untrusted bytecode and produces a well-typedness judgment against a canonical wire format before any operational step fires; like eBPF, the Op verifier is terminating, deterministic, and total. Op differs from JVM in three respects. First, Op's typing judgment includes the compliance-context coordinate κ and the effect row ε; the JVM verifier has neither compliance context nor typed effects. Second, Op's verifier must accept or reject a content-addressed blob, not a classfile with late-bound symbolic references; there is no runtime linkage phase comparable to JVM's constant-pool resolution. Third, Op's verifier must produce a verifier witness (a typed derivation) that downstream replay consumers can check independently, whereas the JVM verifier leaves no persistent artifact. Op differs from eBPF in one respect that is the largest single design choice: eBPF imposes strict termination by bounding all iteration syntactically, while Op relies on the two-axis gas model (structural + extensional) to bound execution via a budget rather than via a syntactic iteration bound. This makes Op more expressive but shifts the verifier obligation from syntactic decidability to budget-compositional soundness. The open research program on verifier fuzz-resistance and malformed-byte handling is the direct Op analogue of the polyhedral static analysis of Gershuni et al. and the end-to-end verification of Nelson et al.

13.6 Effect systems

Typed effect discipline in Op's sense descends from Lucassen and Gifford (POPL 1988), with the algebraic-effect tradition (Plotkin and Pretnar; Bauer and Pretnar 2014) and row-polymorphic effect systems (Koka; Leijen 2014) as the contemporary reference points. Op's effect row is not row-polymorphic; it is a fixed finite vocabulary with a subset order, chosen because admissible compliance effects are bounded by the primitive taxonomy rather than by user-definable signatures. The sanctions-dominance law (a distinguished bottom-propagating effect) diverges from the algebraic-effect tradition: handlers do not recover from `sanctions_check` failure, and the grammar enforces dominance at compile time.

13.7 Sagas, BPEL, and workflow engines

The compensating-transaction discipline originates with Garcia-Molina and Salem (1987): a long-running transaction decomposes into sub-transactions each paired with a compensating action that semantically undoes it. WS-BPEL 2.0 (OASIS, 2007) lifts the discipline into grammar by scoping `<compensationHandler>` to the `<scope>` it inverts, a structural rule Op's `s { body } compensate { inverse }` also adopts. Temporal (Temporal Technologies, 2019-present) and Cadence (Uber, 2017) carry the discipline into production workflow engines, exposing scoped `compensate` as first-class control flow with deterministic replay. Op adopts scoped compensation and composes it with (i) a typed effect row controlling which compensations are required, (ii) a bottom-propagating sanctions-dominance law enforced at the type level, and (iii) cross-zone replay semantics in which compensation receipts are themselves replayable artifacts.

13.8 Move

Linear types originate with Girard's linear logic (1987) and Wadler's "Linear Types Can Change the World" (1990); Rust's affine-ownership discipline is the most influential production realization. Move (Blackshear et al., 2019) synthesizes these into a bytecode-level resource discipline in which values of resource type cannot be copied, discarded, or implicitly duplicated; its contribution is the integration of the discipline with an on-chain asset model.

Op adopts the Move-style integration of linear types with a bytecode-level resource discipline (§4.10, `Linear<T>` and the corridor typestates `Locked<T, ω, ε>`, `Signed<V, ω, ε>`, and `Verified<ω, ε>`). The design decision follows Move's lead: the conservation laws a compliance workflow must satisfy (ownership conservation, resource linearity) are best stated as type-system properties rather than as runtime checks. Op diverges from Move in scope: Move targets on-chain asset management; Op targets off-chain compliance-carrying operations. Move has no suspension, no compensation, and no jurisdiction scoping (its execution environment is a single globally-consistent chain, so these distinctions do not apply). Op's additions to the Move-style linear discipline (the indexed corridor typestates, compensation attachment, and suspension semantics) distinguish Op from a direct port of Move to an off-chain compliance setting.

---

14. Open Problems

14.1 Mechanization of the Operational Semantics

§6's operational semantics are stated informally. A mechanized version, in Coq or Lean, as a small-step reduction relation on a formally defined expression-and-configuration syntax, would discharge §7's conservation invariants as proven theorems rather than proof sketches. The gap between informal statement and mechanized version is a sequence of technical obligations (substitution lemma, progress, preservation, determinism of reduction); the direction is standard, the work nontrivial.

14.2 Preservation for the Full Compilation

§8 states verdict-preservation for the admissible fragment of the rule logic. Extending compilation to the full rule logic, which includes modal operators, temporal coercions, and discretion holes, requires an extended Op target. The discretion-hole target is closest: a hole compiles to an Op `Paused` token pending a PCAuth-witnessed event. Modal and temporal operators have no direct Op targets; their compilation is open.

14.3 Mechanization of Adequacy Directions (b) and (c)

The adequacy schema (§8.6) is supported stepwise in `CompilationSoundness.v` for direction (a). Directions (b) and (c) remain open; §8.6 gives theorem schemas and proof strategies only. Direction (b) would require a coinductive up-to-τ bisimulation argument in the Milner-Sangiorgi style; the natural

mechanization target is the Paco library for coinductive reasoning in Rocq, with the bisimulation relation `R = {(M, [[M]]) | M ∈ L_adm}` as the greatest fixed point. Direction (c) requires formalizing the compliance-carrying judgment `Γ | κ ⊢ e : T ! ε ⇒ κ'` in Rocq, which shares infrastructure with the preservation-and-progress development listed under §14.1.

14.4 Mechanized Confluence Under Parallel Composition

The confluence lemma underlying the parallel-composition rule (§6.6) is sketched on paper by induction on branch-interleaving swaps. Mechanizing the lemma in Rocq or Lean, against the full quintuple configuration and the compensation stack, is open and complements the mechanization of the operational semantics (§14.1).

14.5 Byzantine Liveness Beyond Accountable Safety

The BSC typestate layer closes the safety invariants (I1)-(I3), but full liveness under Byzantine equivocation, prolonged partition, and watcher disagreement remains outside the present proof boundary. Extending the session, typestate, and rely-guarantee story to that setting is still open.

14.6 Cross-Zone Commit Under Byzantine Participants

The corridor type system reifies an imported same-signer/same-index conflict as `Blame<Z, ω, ε>` once the conflict predicate is present, but automatic detection, punishment, or corridor-wide exclusion of a blamed zone remains external to Op. Connecting typed blame witnesses to watcher slashing or governance remediation is outside this paper's scope. §10.8 states bounded blocking before finality and certificate-driven recovery after finality under a fail-stop adversary. Byzantine equivocation requires external mechanisms (the watcher layer from the companion kernel-network paper). The Byzantine-tolerant extension is outside this paper's scope; its correctness obligation is open.

14.7 Verified Primitive Contracts

The Hoare/separation-logic layer reasons about primitives through stated contracts. A full proof that every production primitive and every compensation block implements its contract against live external services remains open, because those services are extensional and evolve independently of the bytecode.

14.8 Recursive and Higher-Order Workflow Fragments

Op rejects unbounded recursion (§6.14). A decidable check for recursion bounded by a static gas limit is standard; a decidable check for recursion bounded by an input-dependent cardinality ("iterate over the members of the participant list, whose length is bounded by an input-dependent constant") requires richer static analysis. This is a standard PL problem; the specific Op instance involves integration with extensional-gas metering. The present calculus is intentionally finitary. Extending the compliance-carrying judgment,

refinement layer, and typestate calculus to richer recursion or user-defined concurrency without losing the current metatheory remains open.

14.9 Compensation Correctness and Extensional Inverses

Op's compensation discipline requires that the declared inverse of a step inverts the step's committed side effect. This is not enforceable at the type-system level because the side effect is extensional (a write to a service whose semantics the Op type system does not fully capture). A conservative static check rejects compensation clauses whose effect row does not subsume the forward's; a complete check requires modeling the external service's state. Section 11 sketches the Hoare/separation-logic layer and VC generation that such a check would use; the open problem is a complete primitive-specification library and a mechanized soundness proof for the generated verification conditions. The open problem has two parts: (a) a specification language for primitives that captures enough of their extensional behavior to permit an extensional-inverse check (Hoare triples per primitive, separation assertions on the pre- and post-footprints); (b) a verification-condition discharge story (SMT-level reasoning with primitive-library lemmas, or Iris-level reasoning with a CMRA for the external service). The present paper states the intensional rule and flags the extensional check as external; the research program is to make the extensional check internal and mechanized.

14.10 Gas Pricing Under Adversarial Cardinality

The extensional-gas model relies on cardinality certificates (§4.9) when compile-time bounds are unavailable. A malicious caller might supply a certificate under-reporting the true cardinality, deliberately exhausting gas mid-execution. The protocol's response (failing closed and running compensation) is correct but wasteful; a pricing model that disincentivizes adversarial cardinality under-reporting is open. The calculus separates structural gas from extensional gas, but the economic calibration of cardinality certificates, long-lived suspensions, and hostile callback timing is still open. The safety story is closed; the pricing story is not.

14.11 Compilation to Zero-Knowledge Circuits

Following the compilation-to-circuits line from the companion rule-logic paper (§11 of that paper), an Op program compiled to an arithmetic circuit would produce a zero-knowledge proof that execution was correct without revealing the inputs. The admissible fragment's finitary nature suggests the compilation is feasible; the primitive-layer wrappers (service calls to external systems) are the hard case because they are not encoded as circuit operations. A reasonable target is a hybrid proof: Op control-flow and data-flow as a circuit, primitive calls as external oracle attestations. Design is open.

14.12 Confidential Proof Transport

Non-interference constrains what low observers can learn from high inputs, but succinct zero-knowledge transport for proof bundles and selective disclosure across zones remain open. The challenge is to compress or hide evidence without losing replayability or typed provenance.

14.13 Memory Model and Transaction-Isolation Semantics

The parallel composition rule treats branches as interleaving independently over disjoint primitive footprints, with confluence provable by commuting-diamond arguments on branch-interleaving swaps. A full memory model, a formal semantics for what a primitive "reads" and "writes" under the ambient service implementation, and what memory orderings an Op program observes under concurrent external state, is outside the present paper. The relaxed-memory model literature (Batty-Sewell C11, CompCertTSO, Lahav-Vafeiadis Promising Semantics) provides the canonical toolkit; transaction-isolation semantics (Adya-Liskov-O'Neil; Fekete-O'Neil-O'Neil) supplies the analogous database framework. The research program is to state Op's memory model formally, prove that the current parallel-composition rule preserves the needed ordering guarantees, and document the isolation level primitives must satisfy.

14.14 Information-Flow Security

The present paper treats sanctions-dominance as a one-bit non-interference property: sanctions-tainted data cannot reach a verdict-ok continuation. The narrow mechanized core is stronger than a prose slogan and weaker than full IFC. `ComplianceContext.v` proves sanctions absorption under context meet and now names the exact context-algebra corollaries `sanctions_bottom_cannot_become_compliant_after_meet` and `sanctions_bottom_forces_rank_zero_after_meet`; `CompilationSoundness.v` closes the sanctions and filled-hole preservation cases, with filled-hole attestation append treated as τ-labelled witness transport; `LexOpAdequacy.v` closes finite verdict agreement modulo `prelude` and `host_sanctions`; and `PCAuthQuorum.v` closes structural quorum facts for accepted PCAuth witnesses. Together these support the theorem schema `SanctionsBottomAndTauAttestationNonBypass`: under a verified PCAuth witness, a filled-hole attestation transported through τ-append cannot convert a sanctions-bottom terminal context into a verdict-ok continuation, because the terminal context is computed by the absorbing meet. This schema is the current one-bit IFC boundary; it still needs to be packaged as a single theorem over the real Op transition system.

A richer IFC treatment introduces a lattice of security labels over jurisdictional, personal-data, and commercial sensitivity classes; a typed declassification discipline for legally authorized downstream release; and a full non-interference theorem against a multi-level attacker. This is a legitimate research program that extends but does not contradict the present bottom-propagation story. Sabelfeld and Myers (2003), Pottier and Simonet (2003), and the FlowML/JFlow line supply the reference technology. The key question is which legal authorizations correspond to which declassification events; this is a domain-modelling problem, not a pure PL problem.

14.15 Game-Semantics Account of Compliance

A Hyland-Ong / Abramsky-Jagadeesan-Malacaria game-semantics account of Op would model the evaluator against an adversarial environment whose moves include sanctions update, consent withdrawal, oracle equivocation, and participant defection. Compliance would be a winning condition on the observable

traces: no trace reaches a forbidden verdict transition under the adversary's moves. A full game-semantics treatment would (a) refactor "adversarial defenses" from a list of local mitigations into a single winning-condition statement, and (b) align Op with the game-semantics treatments of ML effects (Abramsky-McCusker; Hyland-Ong) and of concurrent processes (Honda-Tokoro; Rideau-Winskel). The present paper leaves this as a research program: the adversary taxonomy is stated operationally in the adversarial-defenses section, outside a strategy and outside a formal winning condition.

14.16 JIT and Backend Verification for the Extracted Interpreter

The backend-compilation section specifies an AOT pipeline into WebAssembly. A verified JIT variant, compile-on-demand with per-function caching, deoptimization on cardinality-certificate refutation, and preservation under the observational equivalence of the bisimulation-theory section, is a distinct research program. The JVM HotSpot tier-up semantics (Paleczny, Vick, Click), V8's TurboFan/Ignition pipeline, and the CompCertELF verified-backend line are the technology reference. The open question is whether the two-axis gas model (structural + extensional) admits a JIT discipline in which pre-deoptimization and post-deoptimization traces are bisimilar modulo gas-accounting stutter, and whether this bisimulation can be mechanized.

14.17 Resource-Aware Parametricity and Relational Cost

Op's metatheory is unary: a single well-typed program, a single execution, a single set of conservation invariants. A relational, resource-aware parametricity theorem (in the style of Çiçek-Barthe-Gaboardi RelCost 2017; Radicek-Barthe BiRelCost 2018) would state that any two well-typed Op programs typed at the same type up to a relational refinement satisfy a cost-adjacent relation on their executions, uniformly in gas and in the compliance-context algebra. Such a theorem is the relational analogue of the abstraction theorem for unary parametricity; it would justify compositional cost reasoning for program transformations (inlining, compensation hoisting, effect coalescing) and for contract refinement.

14.18 Nominal vs. Structural Typing Discipline for Corridor Identity

The corridor types `Locked<T, ω, ε>`, `Signed<V, ω, ε>`, and `Verified<ω, ε>` carry an identity-bearing ω index (the corridor identifier) and an ε index (the effect row). Whether this discipline is nominal (corridor identifiers are opaque names; equality by name only) or structural (corridor identifiers are tuples of derived attributes; equality by attribute) is a design question the present paper leaves implicit. The reference treatments are Pierce's *Types and Programming Languages* (nominal vs. structural subtyping), the nominal-logic line (Pitts; Gabbay-Pitts), and the typing-conventions survey (Cardelli; Palsberg). The research program is to make Op's corridor-identity discipline explicit, state the resulting equational theory on corridor types, and derive the appropriate mechanized subsumption rule.

14.19 Verifier Fuzz-Resistance and Malformed-Byte Handling

The bytecode wire-format pins the canonical encoding and the content-addressable hash. The verifier-fuzz discipline, a formal statement of how the verifier behaves under arbitrary byte sequences, including truncations, malformed length prefixes, and cryptographic-format corruption, is an adjacent but distinct research program. The JVM verifier's treatment of malformed classfiles (Stata-Abadi; Freund-Mitchell) and eBPF's in-kernel verifier (Gershuni et al. PLDI 2019; Nelson et al. OSDI 2020) supply the reference methodology. The research program is to state Op's verifier as a total function over byte sequences, with an explicit rejection verdict for every malformed input and a formal bound on verifier runtime as a function of input length.

14.20 PCAuth Transport: Delegation, Expiration, Revocation, Multi-Signer, Caching

The PCAuth verifier persists and re-validates attestations produced inside Lex rule evaluation. The transport contract in the present paper covers value-binding, freshness against the current signing-key epoch, and gas cost. The fuller transport contract, delegated PCAuth issuance from a root signer through a scoped delegated signer (mirroring Lex's delegation tree), expiration with explicit per-witness TTLs, revocation of previously-issued PCAuths under compromise, multi-signer threshold attestations (quorum-of-k-of-n), and deterministic caching under replay, is a transport-protocol research program. The reference technology is X.509 PKIX revocation (CRL, OCSP), WebPKI certificate transparency (Laurie, Langley, Kasper), and the post-quantum signature transport work (NIST PQC: Dilithium, Falcon, SPHINCS+). The present paper states the single-signer transport contract; the delegated, revocable, multi-signer transport discipline is open.

14.21 Multi-Party Session Typing Beyond the Bilateral Corridor

The corridor calculus gives a bilateral session typing (two zones, commit-acknowledgement dance). Multi-party session types in the full MPST sense (Honda-Yoshida-Carbone 2008; Scalas-Yoshida 2019; Deniélou-Yoshida asynchronous MPST) admit n-ary protocols with branching, recursion, and independent subsession composition. An Op extension to n-party corridors, ADGM, DIFC, and MAS all participating in a single signed commitment with coordinator rotation, per-zone veto, and explicit finality witnesses, is a natural research program. The present paper projects n-party protocols onto bilateral pairs and composes them by corridor-graph meet; the full n-party typing story is open.

14.22 Decompilation and Source Maps

The Lex→Op compilation is forward-only: a Lex program is lowered to an Op program, with a compilation rule and a preservation theorem. A reverse translation, an Op program re-expressed as a Lex program, with a source-map discipline that recovers Lex variable names, hole identities, and judgment positions, is a distinct research program. The decompilation case is interesting because Op programs are emitted by auditors and tooling that can discard Lex source. The reference technology is the source-map line for JavaScript (Loth; Google SourceMap v3) and the verified-decompilation work for low-level IRs (Myreen; Peskin-Myreen).

The research program is to state an Op→Lex partial decompilation, define its soundness against the forward compilation, and mechanize the round-trip for the admissible fragment.

14.23 Logical-Framework Adequacy for Lex-to-Op

A logical-framework adequacy proof in the Harper-Honsell-Plotkin sense would give Lex and Op a shared LF encoding and prove that the compilation is adequate with respect to that encoding: every Lex derivation has a unique LF witness, every Op program's typing has a unique LF witness, and the compilation rule extends to an LF-level map whose image is exactly the LF-encodable fragment of Op. This is the standard adequacy obligation for dependent-type-theory compilation projects (the Twelf-mechanized LF work; the Beluga metalanguage work of Pientka); it is not the same as the operational adequacy theorem already stated. The research program is to give the LF encoding, prove compositional adequacy, and mechanize the result.

14.24 Temporal Coherence and Pack-Evolution Lowering

Lex's temporal polarity discipline (past-admissible, future-typed) is internal to the Lex calculus. Pack evolution, the process by which a jurisdiction updates its rule logic and reissues compliance certificates under the new pack version, is external to both Lex and Op, handled at the pack-registry layer of the companion SJN paper. The runtime lowering of a Lex rule whose admissibility was valid under pack $\mathtt{p}_1$ and invalid under pack $\mathtt{p}_2$ is not directly represented in Op's current compilation. An Op evaluator that encounters a proof-bundle entry produced under pack $\mathtt{p}_1$ and evaluates it under pack $\mathtt{p}_2$ checks the pack-version anchor on the entry (§8.3 PCAuth transport) and rejects a mismatch, but the downstream effect of that rejection, what the evaluator should do with an already-committed state that is now under a later pack version, is delegated to the pack-registry layer. The research program is:

1. formalize the pack-evolution transition as a typed relation $\vdash \mathtt{p}_1 \rightarrow \mathtt{p}_2$ on pack versions;
2. extend the Op compilation rule to carry a pack-version annotation on every compiled expression;
3. state a temporal-coherence theorem: for every Op execution trace spanning a pack transition, either the trace is rejected at the pack-version mismatch or the post-transition state is expressible as an admissible state in the later pack.

This is a distinct research program from the behavioral temporal logic (LTL/CTL/TCTL) program; it is specifically about pack version as legal time rather than about workflow time.

14.25 Adversarial-Bytecode Residual Attacks Beyond §5

The adversarial-defenses section (§5) covers five concrete attack classes: currency-index confusion (closed by `Money<curr>`); ensures-to-evidence forgery (closed by paired `EvidenceEntry` introduction); PCAuth replay against a different payload (closed by `input_op(h) = v` value-binding); oracle-response forgery (closed by `AuthOracle_`τ signed-response check); and compliance-primitive impersonation (closed by the

substance-over-form registry-digest rule). Two residual adversarial-bytecode attack classes are not closed here.

Canonical-encoding ambiguity. The bytecode wire format declares itself canonical and content-addressable, but the formal soundness proof of canonicity, that every well-typed Op program has exactly one admissible byte representation, and that every admissible byte representation deserializes to exactly one well-typed program, is mechanized for the core AST (`CanonicalEncoding.v`); extension to the full bytecode surface remains open, including compensation blocks, corridor typestate witnesses, and PCAuth-embedded proof-bundle entries. An attacker who finds a non-canonical encoding for an otherwise well-typed program could break content-addressability. The research program is to extend the mechanized canonicity proof to the full surface.

Authority binding across pack versions. The PCAuth verifier checks a witness against the current signing-key epoch and the current non-revocation state, but it does not currently check that the *pack version* of the rule logic that produced the witness matches the pack version the Op evaluator is running. An attacker who replays a witness from pack version $\texttt{p}_1$ into an Op evaluator running pack version $\texttt{p}_2$ could, in principle, benefit if $\texttt{p}_2$ has narrowed the hole's scope or tightened its predicate while $\texttt{p}_1$ left it looser. The immediate mitigation is the pack-version field in the bulletin-board anchor; the formal mitigation is an explicit pack-version typing obligation at `fill` admission. The research program is to extend the compilation rule so that the witness type carries a pack-version index and the admissibility check rejects pack-mismatch witnesses.

---

15. Backend Compilation

15.1 Target and Ahead-of-Time Pipeline

Op admits an ahead-of-time backend targeting WebAssembly 2.0 (Haas et al., 2017). Write `[[·]]_wasm : Op → Wasm` for the compiler, defined compositionally by `[[·]]_wasm = emit_wasm ∘ lower_ir`. The pipeline is functional:

`Op bytecode → Op IR → Wasm`.

`Op IR` is a first-order control-flow graph with explicit terminals, gas checkpoints, suspension edges, and primitive imports. The first pass makes the bytecode's evaluation order and join structure explicit. The second lowers each IR block to structured WebAssembly functions plus imported host calls for the primitive layer. WebAssembly is therefore the backend artifact, not the source semantics: compensation, sanctions dominance, jurisdiction scope, and typed suspension remain defined by Op and are preserved by lowering.

15.2 Compilation Correctness

Let `T_Op` be the terminal abstraction mapping backend exits to the Op terminal set `Return(v)`, `Paused(E, K_blob, b)`, `Halted(err, mu)`, `SanctionsBlocked(mu)`, `AuthorizationBlocked(mu)`, `OutOfStructuralGas(mu)`, `OutOfCompensationGas(mu)`, and `Timeout(mu)`, forgetting backend-local program counters, Wasm locals, and fuel counters.

Correctness obligation (backend compilation correctness). For every closed, well-typed Op program `P` and input `x` in its declared input type, the target backend theorem is `Op-run(P, x) ≡ T_Op(Wasm-run([[P]]_wasm, x))`. Equivalently, `Op-run(P, x) ≡ Wasm-run([[P]]_wasm, x)` modulo the operational-terminal abstraction `T_Op`.

Proof sketch target. The argument has the CompCert shape (Leroy, 2009): compose two forward simulations. `lower_ir` preserves the Op small-step relation by construction, because it only makes evaluation order, terminals, and gas checkpoints explicit. `emit_wasm` lowers each IR control point to a WebAssembly block with a simulation relation matching the IR environment, proof-bundle accumulator, compensation stack, and gas meters to Wasm locals, linear memory, and imported host state. Pure control-flow steps become finite sequences of deterministic Wasm reductions. Primitive dispatches lower to imports carrying the same preconditions and postconditions as the Op primitives. Suspension and failure terminals lower to tagged exits rather than raw Wasm traps, so every backend terminal reifies to an Op terminal under `T_Op`. Induction on the Op execution derivation yields terminal equivalence; determinism of both semantics collapses the simulation to equality of observable terminals. The proof is not mechanized in the public repository at this time.

15.3 Gas Mapping and Specialization

Structural gas maps to Wasm fuel. Every rule-applying IR node begins with a fuel check and decrement; exhaustion maps to `OutOfStructuralGas`, and the same instrumentation on compensation paths maps compensation exhaustion to `OutOfCompensationGas`. Extensional gas maps to a per-primitive Wasm cost table `C_wasm`: a primitive dispatch `p` with cardinality witness κ consumes `C_wasm(p, κ)` units before the host call.

When the ownership-witness index ω and effect-row index ε are statically known, `Locked<T, ω, ε>` is monomorphized to a fixed-layout record and stack-allocated across straight-line regions, spilling only at suspension boundaries. The same effect-row analysis allows the backend to elide generated sanctions stubs on blocks whose ε is statically disjoint from `sanctions_check`; elision is permitted only when the source typing derivation already proves the checks absent.

15.4 JIT Variant, Size, and Measurement

A second backend `[[·]]_wasm_jit` emits the same AOT-safe WebAssembly plus tracing-JIT hints: hot-trace anchors, branch-frequency counters, deoptimization targets, and inline-cache slots for primitive im-

ports. The design follows trace-based specialization (Gal et al., 2009) and the monomorphic/polymorphic inline-cache discipline used by V8. The hints are semantically inert: a runtime that ignores them still executes the AOT-correct module.

The engineering surface is modest: 3 KLoC for the AOT pipeline, split across the Op-to-IR normalizer, the Wasm emitter, and the fuel/specialization pass. This paper reports no backend benchmark. The right empirical comparison is against EVM implementations on matched workloads; that measurement is future work rather than a claim of the present paper.

---

16. Conclusion

Compliance-carrying operations are a distinct kind of program. They require typed control flow, typed evidence transport, typed cross-zone state transition, and typed failure recovery. Op states those requirements in one language design: a compliance-carrying typing judgment, small-step semantics, BSC-indexed typestates, session-typed corridor protocols, Hoare/separation logic with rely-guarantee reasoning, open information-flow-security schemas, bisimulation obligations for Lex-to-Op compilation, refinement typing with SMT discharge, PCAuth transport, a fixed binary format, and a Wasm backend with explicit correctness targets. Several of those targets remain open mechanization work, as listed in §14.

That matters because institutional workflows usually fail at the seams between configuration, evaluator, and human procedure. Skipped sanctions gates, unreversed compensations, ambiguous ownership, and unstructured cross-zone commits become either ill-typed programs or semantically blocked states rather than operator-discipline problems discovered in audit. The cross-zone letter-of-credit case study in Section 12 shows that the resulting artifact is still readable as workflow and as metatheory.

What remains open is narrower than before but still real. Full Byzantine liveness, complete verified contracts for extensional primitives and compensators, recursive extensions that preserve the current metatheory, and succinct confidential proof transport are not closed here. Those are boundary obligations around the language and its host environment, not uncertainty about the core calculus.

The central claim is therefore modest and precise. A workflow that changes sovereign state should be written in a language strong enough to describe the evidence it carries, the commit protocol it depends on, and the failure modes it refuses to admit. Op is that language-level account.

---

17. Glossary

Term	Precise definition	First introduced
Step	A named, typed unit with input type, output type, effect row, body, optional suspension and compensation.	§4.2
Primitive	A kernel-recognized action with stable lowering rules (`create.entity`, `consent.board_resolution`, etc.).	§4.3
Program	One executable operation family, identified by a stable operation identity in the program envelope.	§4.1
Effect row	A set of effects in the typing judgment Γ `|` $\kappa \vdash$ `e : T !` ρ.	§4.10
Effect	A kernel-tracked operational capability (`sovereign_write`, `identity_mutation`, `fiscal_transfer`, `sanctions_check`, `governance_request`, `document_generation`, `external_read`, `proof_emit`, `await`). Distinct from Lex-paper effects.	§4.10
Contract	A declaration `requires`/`ensures` on a step, stating preconditions and postconditions over compliance domains.	§4.7
Compliance domain	One axis of the bytecode-domain vocabulary, either from the 23-module standard library or from a pack-defined extension. Distinct from "domain" used in type theory.	§4.8

Term	Precise definition	First introduced
Jurisdiction (ambient)	The jurisdiction declared by the operation envelope at lowering.	§4.5
Jurisdiction (scoped)	An `in <juris> { ... }` block that rebinds the ambient jurisdiction for a program region.	§4.5
Zone	Operational synonym for a sovereign deployment under one jurisdiction identifier. Not a geographic area.	§10
Authority (Op)	A string field on `RegulatoryApproval { authority }` naming the regulator. Distinct from Lex tribunal.	§4.6
Compensation	A typed inverse program attached to the step it inverts. Reverse-topological, idempotent, best-effort.	§2.2
Session type	A typed protocol describing the message/resource flow at an endpoint.	§10.1
Linear resource	A value consumed exactly once under the linearity discipline.	§2.4
Locked resource	A resource in the `Locked<T, ω, ε>` typestate, consumed by `commit_transfer` or `release_lock`.	§2.4
Signed witness	A locally unique verdict in the `Signed<V, ω, ε>` typestate.	§2.4
Verified witness	A durable bilateral verdict pair in the `Verified<ω, ε>` typestate.	§2.4

References

Aldrich, J., Sunshine, J., Saini, D., and Sparks, Z. (2009). "Typestate-Oriented Programming." In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object-Oriented Programming Systems Languages and Applications (OOPSLA 2009)*, pp. 1015-1022. ACM.

Ahmed, A. (2006). "Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types." In *Proceedings of the 15th European Symposium on Programming (ESOP 2006)*, pp. 69-83. Springer LNCS 3924.

Allombert, V., Bourgoin, M., and Tesson, J. (2019). "Introduction to the Tezos Blockchain." In *Proceedings of the 17th International Conference on High Performance Computing and Simulation (HPCS 2019)*. IEEE. arXiv:1909.08458.

Bauer, A. and Pretnar, M. (2014). "An Effect System for Algebraic Effects and Handlers." *Logical Methods in Computer Science*, 10(4).

Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., and Zanella-Béguelin, S. (2016). "Formal Verification of Smart Contracts: Short Paper." In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS 2016)*. ACM.

Boneh, D., Drijvers, M., and Neven, G. (2018). "Compact Multi-Signatures for Smaller Blockchains." In *Advances in Cryptology: ASIACRYPT 2018*. Springer.

Blackshear, S., Cheng, E., Dill, D. L., Gao, V., Maurer, B., Nowacki, T., Pott, A., Qadeer, S., Rain, Russi, D., Sezer, S., Zakian, T., and Zhou, R. (2019). "Move: A Language with Programmable Resources." Diem Association.

Brookes, S. (2004). "A Semantics for Concurrent Separation Logic." In *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR 2004)*, pp. 16-34. Springer LNCS 3170.

Buterin, V. (2016). "EIP-150: Gas Cost Changes for IO-Heavy Operations." Ethereum Improvement Proposal. https://eips.ethereum.org/EIPS/eip-150

Caires, L. and Pfenning, F. (2010). "Session Types as Intuitionistic Linear Propositions." In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, pp. 222-236. Springer LNCS 6269.

Castro-Perez, D., Seco, J. C., and Vasconcelos, V. T. (2020). "Polymorphic Session Types." *PACMPL* 4(POPL), Article 69, pp. 1-30.

Coppo, M., Dezani-Ciancaglini, M., Padovani, L., and Yoshida, N. (2015). "A Gentle Introduction to Multiparty Asynchronous Session Types." In *Formal Methods for Multicore Programming*, pp. 29-66. Springer LNCS 9105.

Dardha, O. and Gay, S. (2018). "A New Linear Logic for Deadlock-Free Session-Typed Processes." In *Foundations of Software Science and Computation Structures (FoSSaCS 2018)*, pp. 91-109. Springer LNCS 10803.

Denning, D. E. (1976). "A Lattice Model of Secure Information Flow." *Communications of the ACM*, 19(5), 236-243.

Dwork, C., Lynch, N., and Stockmeyer, L. (1988). "Consensus in the Presence of Partial Synchrony." *Journal of the ACM*, 35(2), 288-323.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). "Impossibility of Distributed Consensus with One Faulty Process." *Journal of the ACM*, 32(2), 374-382.

Foster, J. S., Terauchi, T., and Aiken, A. (2002). "Flow-Sensitive Type Qualifiers." In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI 2002)*, pp. 1-12. ACM.

Freund, S. N. and Mitchell, J. C. (2003). "A Type System for the Java Bytecode Language and Verifier." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(6), 793-832.

Gal, A., Probst, C. W., and Franz, M. (2009). "Trace-based Just-in-Time Type Specialization." In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009)*. ACM.

Garcia-Molina, H. and Salem, K. (1987). "Sagas." In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*. ACM.

Girard, J.-Y. (1987). "Linear Logic." *Theoretical Computer Science*, 50(1), 1-101.

Gray, J. and Lamport, L. (2006). "Consensus on Transaction Commit." *ACM Transactions on Database Systems*, 31(1), 133-160.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. F. (2017). "Bringing the Web up to Speed with WebAssembly." In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM.

Haber, S. and Stornetta, W. S. (1991). "How to Time-Stamp a Digital Document." *Journal of Cryptology*, 3(2), 99-111.

Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., and Rosu, G. (2018). "KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine." In *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF 2018)*. IEEE.

Honda, K., Vasconcelos, V. T., and Kubo, M. (1998). "Language Primitives and Type Discipline for Structured Communication-Based Programming." In *Programming Languages and Systems (ESOP 1998)*, pp. 122-138. LNCS 1381.

Honda, K., Yoshida, N., and Carbone, M. (2008). "Multiparty Asynchronous Session Types." In *Proceedings of the 35th Symposium on Principles of Programming Languages (POPL 2008)*, pp. 273-284. ACM.

Honda, K., Yoshida, N., and Carbone, M. (2016). "Multiparty Asynchronous Session Types." *Journal of the ACM*, 63(1), Article 9, pp. 9:1-9:67.

Jones, C. B. (1983). "Specification and Design of (Parallel) Programs." In *Proceedings of the IFIP 9th World Congress*, pp. 321-332. North-Holland.

Kildall, G. A. (1973). "A Unified Approach to Global Program Optimization." In *Proceedings of the 1st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1973)*, pp. 194-206. ACM.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. (2009). "seL4: Formal Verification of an OS Kernel." In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)*. ACM.

Jung, R., Krebbers, R., Jourdan, J.-H., Bizjak, A., Birkedal, L., and Dreyer, D. (2018). "Iris from the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic." *Journal of Functional Programming*, 28, e20.

Kumar, R., Myreen, M. O., Norrish, M., and Owens, S. (2014). "CakeML: A Verified Implementation of ML." In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014)*. ACM.

Leijen, D. (2014). "Koka: Programming with Row Polymorphic Effect Types." In *Proceedings of the 5th Workshop on Mathematically Structured Functional Programming (MSFP 2014)*.

Leroy, X. (2006). "Formal Certification of a Compiler Back-End, or: Programming a Compiler with a Proof Assistant." In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*. ACM.

Leroy, X. (2009). "Formal Verification of a Realistic Compiler." *Communications of the ACM*, 52(7), 107-115.

Lindholm, T., Yellin, F., Bracha, G., and Buckley, A. (2015). *The Java Virtual Machine Specification, Java SE 8 Edition*. Oracle America, Inc.

Lorgat, R. (2025). "Lex: A Logic for Jurisdictional Rules." Companion paper, Momentum Research programme. Available at research.momentum.inc/papers/lex.html.

Lucassen, J. M. and Gifford, D. K. (1988). "Polymorphic Effect Systems." In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1988)*. ACM.

Milner, R. (1980). *A Calculus of Communicating Systems.* Springer LNCS 92.

Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., and Nystrom, N. (2001). "Jif: Java Information Flow." Cornell University. https://www.cs.cornell.edu/jif/

Necula, G. C. (1997). "Proof-Carrying Code." In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1997)*, pp. 106-119. ACM.

O'Hearn, P., Reynolds, J., and Yang, H. (2001). "Local Reasoning about Programs that Alter Data Structures." In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL 2001)*, pp. 1-19. Springer LNCS 2142.

OASIS (2007). *Web Services Business Process Execution Language (WS-BPEL) Version 2.0.* OASIS Standard.

Padovani, L., Vasconcelos, V. T., and Vieira, H. T. (2014). "Typing Liveness in Multiparty Communicating Systems." In *Coordination Models and Languages (COORDINATION 2014)*, pp. 147-162. Springer LNCS 8459.

Park, D. (1981). "Concurrency and Automata on Infinite Sequences." In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pp. 167-183. Springer LNCS 104.

Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.

Pilkiewicz, A. and Pottier, F. (2011). "The Essence of Monotonic State." In *Proceedings of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2011)*, pp. 73-86. ACM.

Plotkin, G. D. (1977). "LCF Considered as a Programming Language." *Theoretical Computer Science*, 5(3), 223-255. Originating statement of the operational-adequacy obligation relating a denotational model to an operational semantics by structural induction on reductions.

Plotkin, G. and Pretnar, M. (2013). "Handlers of Algebraic Effects." *Logical Methods in Computer Science*, 9(4:23). (Preliminary version: ESOP 2009.)

Pottier, F. and Simonet, V. (2003). "Information Flow Inference for ML." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(1), 117-158.

Pous, D. and Sangiorgi, D. (2007). "Enhancements of the Bisimulation Proof Method." Chapter in *Advanced Topics in Bisimulation and Coinduction*, Cambridge Tracts in Theoretical Computer Science 52. Cambridge University Press, 2012. Preliminary version: *Theoretical Computer Science* 373(3), 190-213, 2007.

Reynolds, J. C. (2002). "Separation Logic: A Logic for Shared Mutable Data Structures." In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pp. 55-74. IEEE.

Sabelfeld, A. and Myers, A. C. (2003). "Language-Based Information-Flow Security." *IEEE Journal on Selected Areas in Communications*, 21(1), 5-19.

Sabelfeld, A. and Sands, D. (2005). "Declassification: Dimensions and Principles." In *Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW 2005)*, pp. 255-269. IEEE.

Sangiorgi, D. (1998). "On the Bisimulation Proof Method." *Mathematical Structures in Computer Science*, 8(5), 447-479. Up-to-τ coinduction as a sound bisimulation proof technique; basis for the completeness direction of the adequacy theorem.

Sangiorgi, D. (2011). *Introduction to Bisimulation and Coinduction.* Cambridge University Press.

Scalas, A. and Yoshida, N. (2019). "Less is More: Multiparty Session Types Revisited." *Proceedings of the ACM on Programming Languages* 3(POPL), Article 30, pp. 1-29.

Sevcik, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., and Sewell, P. (2013). "CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency." *Journal of the ACM*, 60(3), 22:1-22:50.

Skeen, D. (1981). "Nonblocking Commit Protocols." In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*. ACM.

Stata, R. and Abadi, M. (1998). "A Type System for Java Bytecode Subroutines." In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1998)*, pp. 149-160. ACM.

Strom, R. E. and Yemini, S. (1986). "Typestate: A Programming Language Concept for Enhancing Software Reliability." *IEEE Transactions on Software Engineering*, SE-12(1), 157-171.

Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.-Y., Kohlweiss, M., Zinzindohoue, J. K., and Zanella-Béguelin, S. (2016). "Dependent Types and Multi-Monadic Effects in F." *In* Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. ACM.

Temporal Technologies (2019-present). Temporal Workflow Engine. Open-source project documentation at https://docs.temporal.io (accessed April 2026).

Uber Technologies (2017). Cadence Workflow Engine. Open-source project at https://cadenceworkflow.io (accessed April 2026).

Wadler, P. (1990). "Linear Types Can Change the World!" In *Programming Concepts and Methods*, M. Broy and C. Jones (eds.). North-Holland.

Wadler, P. (2012). "Propositions as Sessions." In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*, pp. 273-286. ACM.

Wood, G. (2014). "Ethereum: A Secure Decentralised Generalised Transaction Ledger." Ethereum Yellow Paper. https://ethereum.github.io/yellowpaper/paper.pdf (Berlin revision, 2021).