# Lex: A Logic for Jurisdictional Rules

Raeez Lorgat

April 2026

## Lex: A Logic for Jurisdictional Rules

Author: Raeez Lorgat

### Abstract

Lex is a dependently typed logic of jurisdictional rules. Tribunal modals are typed over authority bridges: a canonical bridge `b : CanonBridge(T1,T2,A)` acts covariantly by `coerce[b] : [T1]A → [T2]A`; the canonical strict function-bridge target is Qed-closed in Rocq, while proof-relevant raw bridge strictification, bridge admission, presheaf naturality, and adequacy remain open obligations rather than hidden assumptions. Defeasibility ranges over a priority graph (a directed acyclic graph ordering rules by legal precedence) and evaluates in a five-element Heyting verdict lattice (a distributive lattice equipped with implication, where a rule's verdict is the least upper bound of the outcomes its fired exceptions admit). Typed discretion holes carry PCAuth witnesses (proof-carrying authorization: cryptographic attestations signing the authority, the hole, the exact value supplied, the fill request, and the pack/context digest); structural multi-signer quorum extraction is Qed-closed, while cryptographic unforgeability, credential issuance, bulk verification, revocation transport, and cross-zone bridge validation remain verifier obligations. Temporal stratification separates historical fact at sort `Time_0` from legal consequence at sort `Time_1` through `EffectiveDate`, `Repeal`, stacked `Toll`, and pointwise temporal proposition formers for admissibility-time claims. Behavioral LTL/CTL/TCTL over Op execution traces, including workflow-scale `until`, remains an open Lex-to-Op obligation. A partial Curry-Howard reading (the correspondence between proofs and programs) is available for the recursion-free constructive fragment over finite enumerations, where derivations may be read as proof terms, filled holes as authorized introductions, and tribunal or temporal indices as proposition-carrying modalities; extension to the full calculus is open and discussed in §13.2. The paper specifies a conditional CwF/presheaf semantic target, subject to bridge strictification, bridge-action naturality, and adequacy. Qed-closed formal core now covers

temporal non-regression, pack re-evaluation source preservation, receipt locality, bounded-quantifier admissible expansion, priority normalized-meet agreement, structural PCAuth quorum extraction, a finite-observation event-union support lemma for discretion-hole reductions, canonical strict function-bridge coherence, administrative WHNF reduction, and a fail-closed admission envelope. Full admissible-fragment type-checking still depends on the remaining full-calculus metatheory, but the administrative WHNF kernel `administrative_whnf_bounded_reduction` is Qed-closed; sixteen entries in the mechanization ledger are Qed-closed (including support-kernel statements whose stronger full obligations remain open; see the Mechanization Status appendix at the end of §5.1 for the per-statement enumeration and `lex/formal/coq/Lex/PaperMechanization.v` for the source). Worked cases close the loop on BVI director residency, ADGM-DIFC mutual recognition, Pakistan-directors' tolling, and the ADGM fit-and-proper discretion-hole lifecycle.

---

## 1. The Problem with Compliance-as-Code

Compliance rules are programs. Every financial institution, every RegTech platform, every smart contract that encodes a legal obligation executes formalized law. The formalization runs in Python, Java, SQL, and Solidity. The question is whether the programming language is honest about what it computes.

Existing formalizations share four structural deficiencies.

**Defeasibility simulated by nesting.** A rule that holds "unless" an exception applies, "unless" the exception is overridden, becomes nested if-else branches. Nesting conflates two distinct legal principles: *lex specialis derogat legi generali*, which orders within a rule's exception structure, and *lex posterior derogat legi priori*, which orders across time. Control flow collapses both into one axis. A priority ordering becomes indistinguishable from a temporal supersession without reading every branch. Adding an exception requires touching existing branches, and the semantic independence of the original rules is lost.

**Time treated uniformly.** A variable has one value. A compliance rule has a temporal envelope: an effective date, possible amendments, possible retroactive application. When a statute of limitations is tolled, the original deadline does not vanish; a derived deadline comes into existence alongside it. The original is frozen historical fact. The derived is legal consequence. Conventional programs model this with mutation, conflating "when did this happen?" (no legal system revises the past) with "what are the legal consequences of what happened?" (retroactive amendments routinely revise this).

**Authority invisible.** Identical rules applied to identical facts produce different verdicts under different interpreting authorities. A domestic regulator and a treaty body may disagree whether a beneficial-ownership threshold triggers disclosure. The disagreement reflects the multi-layered structure of legal authority. Programs encode one answer. They carry no mechanism for indexing conclusions by the authority asserting them, or for explicitly bridging authorities under mutual recognition.

**Judgment boundary unmarked.** Determinations such as "fit and proper person," "material adverse change," or "good cause" for a filing extension require human judgment. Existing formalizations hardcode a boolean, insert a TODO comment, or approximate by heuristic. The boundary between derivation and assumption is invisible to downstream consumers. The code cannot distinguish what it computed from what it guessed.

A multi-jurisdictional evaluator needs a language that states which authority asserted a verdict, which exception defeated which rule, which legal consequence was derived from frozen history, and where mechanical evaluation must stop because only an authorized person may decide. Lex is that language.

**Contributions.**

1. Tribunal modals carry a covariant bridge action from authority-recognition witnesses into the semantic universe, with term-level `Tribunal-Assert` and an authority-recognition judgment realizing cross-tribunal transport. The canonical strict function-bridge target is Qed-closed; proof-relevant raw bridge syntax and presheaf naturality remain explicit obligations.
2. Defeasibility separates priority from control flow: rules order on a priority graph and evaluate in a Heyting verdict lattice. Overrides and residual obligations are explicit.
3. Typed discretion holes carry value-indexed PCAuth witnesses with multi-signer quorum, revocation metadata, linked timestamp anchoring, and bounded delegation. A filled hole records who signed, under what authority, in what scope, and against what temporal conditions; the structural quorum-extraction theorem is Qed-closed in Rocq.
4. Temporal stratification is first-class through `EffectiveDate`, `Repeal`, stacked `Toll`, and pointwise admissibility-time temporal proposition formers. Statutory change is typed structure; temporal non-regression and re-evaluation source preservation are Qed-closed over the object-language grade and pack-witness cores.
5. A partial Curry-Howard reading for the recursion-free constructive finite-enumeration fragment: derivations are proof terms, filled holes are authorized introductions, and tribunal and temporal indices delimit the propositions those proofs inhabit. The full-calculus reading is open.
6. A conditional categorical semantic target in a category with families and a presheaf model over bridge contexts, with strictification, bridge-action naturality, adequacy, and full abstraction left as explicit obligations.
7. A Qed-closed receipt algebra for admission hosts: composing Lex receipts by verdict meet and obligation/frontier union is associative, has the expected greatest-lower-bound property on verdicts, and satisfies receipt locality (`accepted/compliant` compose iff each component does).
8. An admissible fragment whose full type-checking theorem is conditional on the remaining full-calculus metatheory, with an administrative WHNF kernel Qed-closed and sixteen mechanization-ledger entries Qed-closed in Rocq, including support kernels whose stronger full obligations remain open (Mechanization Status, end of §5.1).

Section 7 closes the loop on concrete statutory patterns: BVI director residency (defeasibility), ADGM-DIFC mutual recognition (tribunal transport), the ADGM fit-and-proper discretion-hole lifecycle (PCAuth quorum + precedent), and Pakistan-directors tolling (stacked `Toll` over `Time_0`).

---

## 2. What Kind of Law is Computable?

H.L.A. Hart's *The Concept of Law* (1961) established that every legal rule exhibits open texture: a core of settled meaning surrounded by a penumbra of genuine uncertainty. "No vehicles in the park" prohibits a truck; whether it prohibits a motorized wheelchair or a decommissioned tank on a plinth as a memorial is indeterminate as a matter of the rule itself. The point is ontological. The penumbra survives careful drafting because it is a structural feature of general language applied to a world of infinite particularity.

Dworkin's *Taking Rights Seriously* (1977) and *Law's Empire* (1986) argued that law is not exhausted by rules. Legal systems contain principles ("no one should profit from their own wrong") that do not have enumerable exceptions; they have *weight*. How much pull a principle exerts in a given case is a matter of judicial interpretation. Constitutional law, common law reasoning, and interpretive jurisprudence resist formalization because the phenomena are constituted through interpretation.

These observations are correct. Constitutional law, common law reasoning, and jurisprudential questions about the nature of legal validity are not computation. No programming language captures the interpretive courage that reads "equal protection" differently in 1954 than in 1896 without changing the text.

A separate stratum of law is computation. Section 66 of the Seychelles International Business Companies Act: "A company shall have at least one director who is a natural person." This is a predicate over a registered company producing one of two values {`Compliant`, `NonCompliant`}. Evaluation requires two facts. There is no open texture, no principle-balancing. The statute is a program in English.

"File your annual return within 28 days of your fiscal year end" is a function from a date to a deadline. It has an exception for bankruptcy. The exception has an exception for when the bankruptcy court orders otherwise. Defeasible computation, completely mechanical.

Administrative and regulatory compliance rules, the subset governing whether a specific institutional action is permitted, are computation once the formalization is honest about its boundaries. Statutes enumerate their own relevant fact spaces. The Pakistan Companies Act does not require knowing the weather in Karachi; it requires knowing whether the company has a registered office service provider and whether its directors meet the statutory qualifications. The type signature makes the reduction explicit.

Honesty is structural: typed holes distinguish three states. Within the core of settled meaning the evaluator encounters a `MechanicalHole` and discharges it by ordinary typing and reduction. Where the administrative rule reaches an open-textured determination ("fit and proper person," "adequate systems and controls")

and a competent authority exists to answer, evaluation halts at a `DiscretionHole`: a human of specified authority supplies a judgment of specified type. Where the law has no disposition and no officer may lawfully invent one, Lex marks an `UnsettledHole` that clears only by a rule-pack rewrite issued through legislative or regulatory action. Hart's penumbra and Dworkin's harder cases are preserved, separated, and typed.

---

## 3. Five Properties of Honest Compliance Rules

Lex is designed around five properties, each addressing one of the structural deficiencies cataloged above.

### 3.1 Defeasibility

A **defeasible rule** has a base conclusion that holds unless a higher-priority exception applies. In Lex, exceptions are not control flow. They are independently meaningful rules with explicit numeric priority:

```
defeasible
  lambda (ctx : IncorporationContext).
    match ctx.fsra_authorization_required return ComplianceVerdict with
    | False => Compliant
    | True => match ctx.fsra_authorization_status return ComplianceVerdict with
      | FullLicense => Compliant
      | InPrincipleApproval => Pending
      | _ => NonCompliant
  priority 0
  unless
    lambda (ctx : IncorporationContext).
      match ctx.regulated_activity_exemption return Bool with
      | True => True
      | _ => False
    priority 1
end
```

This encodes a rule of the type found in the ADGM Financial Services and Markets Regulations 2015 (FSMR, s.28). The base rule (priority 0) requires FSRA authorization. The exception (priority 1) exempts entities with a regulatory exemption. Priority is explicit, not positional.

Evaluation resolves defeasibility by evaluating exception guards in descending priority order. The highest-priority satisfied exception determines the outcome. If no exception fires, the base body produces the verdict. This directly encodes *lex specialis*: the exception at priority 1 defeats the general rule at priority 0. *Lex*

*posterior* is handled by temporal stratification (Section 3.2), not by defeasibility, the two principles operate on independent axes and must not be conflated.

In the core calculus (Section 4), a defeasible rule is a term with a base type, base body, and a list of exceptions, each carrying a guard, a body, and a priority. The typing rule requires all exception bodies to inhabit the same type as the base body, ensuring that defeasibility does not change the type of the result.

### 3.2 Temporal Stratification

Lex distinguishes two temporal sorts as first-class elements of the type system:

- **Time_0** (frozen historical time): the time at which a transition was committed. "The company was incorporated on 2024-03-15." This is immutable. No legal system retroactively changes when something happened.
- **Time_1** (derived legal time): time produced by a legal rewrite. "The statute of limitations is tolled until 2025-06-01." The original deadline (Time_0) still exists. A new deadline (Time_1) was derived from it by a legal operation (tolling).

The critical invariant is directional: `lift_0` coerces Time_0 into Time_1 (frozen facts can produce derived consequences), but no constructor coerces Time_1 back to Time_0. The mechanized result is a rule-graph theorem about the temporal constructors named in this paper, not a completed embedding theorem for the entire `has_type` relation. Lex carries a typing judgment `G ⊢ t : Time_i` whose intended `i = 0` derivability is closed under exactly the Time_0 introduction forms (date literals, `EffectiveDate`, the Time_0 component of bulletin stamps and PCAuth timestamps). The closed theorem is that the temporal rule graph contains no Time_0-introduction edge with a Time_1 premise and no generated coercion path from `Time_1` to `Time_0`:

**Theorem (temporal rule-graph non-regression).** In the object-language temporal constructor graph generated by `Lift`, `Derive`, `Toll`, `EffectiveDate`, bulletin timestamps, PCAuth timestamps, and pack re-evaluation, no rule with conclusion `Time_0` has a premise of grade `Time_1`, and the reflexive-transitive closure of temporal coercions contains no path `Time_1 → Time_0`.

*Proof.* By finite inspection of the temporal rule grammar. The introduction forms for Time_0 are: date literals (no temporal premises), `EffectiveDate(r, t_0)` whose only temporal premise is `t_0 : Time_0`, and the `Time_0` component of `BulletinStamp` and `PCAuth.timestamp` records (also of `Time_0` premise type). The metalevel projection `src_0(derive_1(t_0, w))` returns the original source fact, but it is not an object-language eliminator from arbitrary `Time_1` terms; it is a projection from the derived-time closure used in the metatheory. No Time_0-introduction rule has a Time_1 premise. The Rocq mechanization is `lex/formal/coq/Lex/TemporalStratification.v`: `no_time1_premise_to_time0_rule` proves that no temporal rule with Time_0 conclusion has a Time_1 premise; `temporal_non_regression` lifts this over temporal derivation trees; and

`no_temporal_retract` proves that the generated temporal coercion graph has no path from `Time_1` back to `Time_0`. `Print Assumptions temporal_non_regression` and `Print Assumptions no_temporal_retract` both report closed under the global context. The remaining full-typing coverage theorem, from the complete `has_type` relation into this temporal rule graph, is a metatheoretic integration obligation rather than something smuggled into this theorem.

Equivalently put: there is no rule of the form `(G ⊢ d : Time_1) → (G ⊢ f(d) : Time_0)` anywhere in the typing rules of §4. A consumer constructing a `Time_1 → Time_0` term outside the parser would have to write a typing derivation for it; no such derivation exists. The typing rules themselves enforce the invariant. The diagnostic message ("there is no coercion from Time_1 to Time_0") issued by the type checker at a misuse site is the operational projection of this theorem.

This models a real legal invariant. Retroactive legislation changes the *consequences* of past events, not the *occurrence* of past events. The Companies Act can be amended to change the filing deadline for companies incorporated in 2024. It cannot be amended to make those companies not have been incorporated. Time_0 records what happened. Time_1 records what the law says about what happened. The two are related by explicit forward coercion and by metalevel derived-time closures, but never by an object-language demotion.

Effective dates, repeal, tolling, and rule-pack evolution are therefore terms, not metadata. The temporal layer gives them explicit types: `EffectiveDate(r, t_0) : Time_0`, `Pack(id, version, rules, effective_date) : PackType`, `Rewrite(P → P') : RewriteWitness`, `Repeal(rule_ref) : RepealWitness`, and `Toll(d, t_tol) : Time_1`. A rule may be evaluated for an event at time `t_e` only when its effective date is no later than `t_e`; repeal has the same temporal shape, except that the rule is removed from the active pack after the repeal date rather than producing a verdict directly.

**Retroactive invalidation and repeal.** The stratification bears directly on retroactive invalidation, where an authority strikes down or reinterprets a rule with effect backward in time. The paradigm example is *Schrems II* (Court of Justice of the European Union, C-311/18, 2020), in which the EU-U.S. Privacy Shield framework was invalidated with immediate effect for all transfers then in progress. An entity that had relied on Privacy Shield certification as its basis for a data-privacy verdict on 2020-07-01 could, the following week, be evaluated against a legal framework in which Privacy Shield never provided a valid basis. A naive compliance system conflates these two evaluations.

The current paper types that transition directly. Let `P_shield` be the pre-invalidation pack and `P_schrems` the post-invalidation pack. The invalidation is a typed witness `Rewrite(P_shield → P_schrems)` induced by `Repeal(privacy_shield_basis)` with its own effective date. A verdict already derived under `P_shield` remains a well-typed `Time_1` object: it is a record of what the law said under that pack. A fresh evaluation of an event after the repeal date, or an explicit `reevaluate(-, Rewrite(P_shield → P_schrems))`, uses the rewritten pack. The weak claim remains syntactic, no constructor exists that coerces a `Time_1` value back to `Time_0`, but the strong claim that derived consequences can be recomputed under a new pack is now internal to the typed temporal layer.

**Stacked tolling.** Tolling is likewise a term, not a comment attached to a deadline. If `d : Time_1` and `t_tol : Time_0`, then `Toll(d, t_tol) : Time_1`. Because the constructor consumes a `Time_1` deadline rather than a `Time_0` fact, tolling stacks: `Toll(Toll(d, t_1), t_2) : Time_1`. The second toll is not a return to historical time. It is a further legal rewrite of an already-derived deadline.

**Authority-relative temporal divergence.** Temporal change is authority-indexed. If authority `A` moves from pack `P_A^0` to `P_A^1` at `t = 100` while authority `B` does not move from `P_B^0` to `P_B^1` until `t = 120`, then an event `e` at `t = 110` may simultaneously inhabit `[A] v_A` and `[B] v_B` with `v_A` $\neq$ `v_B`. This is not inconsistency: the verdicts live under different tribunal modals. Reconciliation, where it exists, requires an explicit bridge witness carrying one verdict across the authority boundary. Where no bridge witness exists, the divergence is a first-class obstruction.

**Temporal obligations.** Lex includes pointwise temporal proposition formers for admissibility-time claims: `$\Diamond \varphi$` (eventually `$\varphi$`) and `$\Box \varphi$` (always `$\varphi$`) over the finite rule-evaluation horizon. Workflow-scale temporal logic over execution traces is not closed here. In particular, behavioral LTL/CTL/TCTL obligations, including a full `until` connective over Op traces, are open at the Lex-to-Op boundary. Lex keeps quantitative time bounds in typed time objects and effective-date guards rather than treating temporal logic as a solved workflow semantics.

Two claims must be separated. The *weak claim*, discharged within the calculus, is that historical facts cannot be overwritten by later legal rewrites. This is syntactic: no constructor exists that coerces a `Time_1` value back to `Time_0`, so a retroactive legal change can alter derived legal consequences while preserving the record of what happened. The stronger claim, that derived consequences can be recomputed under pack evolution, can be stated inside the metatheory once rule packs and rewrite witnesses are made typed objects. The next subsection gives that formalization. What remains open is the full proof that the pack-evolution construction commutes with the modal and temporal subsystem without introducing a hidden `Time_1` $\rightarrow$ `Time_0` path.

### 3.3 Pack Evolution

Temporal stratification prevents later law from rewriting frozen history. A second invariant is needed when the law itself changes: the rule pack against which a derivation was produced must be explicit enough that the derivation can be replayed under a later pack without inventing new historical facts.

When typing or evaluation depends on a particular pack, write `G` $\vdash$`P e : A` for the ordinary Lex judgment relative to the rule constants supplied by `P.rules`. The pack itself is the typed object

```
Pack(id, version, rules : Set Rule, effective_date : Time_0) : Type_0
```

with `id : PackId`, `version : N`, `rules : Set Rule`, and `effective_date : Time_0`. `Pack` is therefore a versioned snapshot of the statutory or regulatory corpus together with the historical date from which that snapshot is legally live.

**Pack digest and content-addressing.** `PackId` is the content-addressed digest of the pack's defining content: `id = digest(canonical_encoding(rules) ++ effective_date)` under a fixed collision-resistant hash (the canonical encoding is the pack layer's analogue of the proof-bundle serialization in §8 of the companion Op paper). Two packs with the same `id` are the same pack as far as Lex is concerned; two packs with the same `(rules, effective_date)` necessarily share `id`. The `version : N` field is an authority-assigned monotone label carried alongside the digest for human-readable accounting; it is not the pack's identity. Rule-identity inside a pack is likewise digest-based: `r in P.rules` is a membership test on a finite set of rule digests, so pack-compatibility is decidable whenever a candidate `omega : Rewrite(P → P')` can be checked against a canonical rule encoding. The admissible-fragment prelude of Section 6 supplies that encoding. The Lex-side pack digest and the pack digest the companion Op paper commits to in its proof bundle (Op §3.5, §4.3, §8) are the same artifact, viewed from the rule-logic layer and the operational-semantics layer respectively.

A pack rewrite is a typed transformation

```
omega : Rewrite(P → P') : RewriteWitness
```

It carries four proof obligations.

- **Conservativity.** Every rule in `P.rules` is either still present in `P'.rules` or comes with an explicit `Repeal(r)` witness. New rules may be added freely.
- **Type preservation.** If a rule `r` is active in both packs, then `type_P(r) = type_P'(r)`.
- **Effective-date monotonicity.** `P'.effective_date ⩾ P.effective_date`.
- **Replay transport.** For every derived-time witness that the rewrite claims to preserve, the rewrite carries a function transporting a `legal_witness P` into a `legal_witness P'`. If a cited rule is repealed and no replacement proof exists, no replay witness exists for that derivation; re-evaluation is then blocked rather than silently coerced.

The witness is typed because a syntactic patch to the rule text is not enough; one must also prove that the pack boundary moved forward in time and that surviving rules did not silently change type.

Write `w_P` for a derivation witness whose cited rules all lie in `P.rules` and whose legal effect is valid no earlier than `P.effective_date`. The derived legal time `derive_1(t_0, P, w_P)` therefore records the frozen fact `t_0 : Time_0`, the pack under which the consequence was derived, and the pack-relative witness that justifies the consequence.

The re-evaluation operator is the metalevel map

```
reevaluate : DerivedTime(P) x Rewrite(P → P') → DerivedTime(P')
```

defined by

```
reevaluate(derive_1(t_0, P, w_P), omega) :=
  derive_1(t_0, P', replay_witness(omega, w_P))
```

where `w_P'` is obtained by replaying the same `Time_0` witness through the active rules of `P'` justified by `omega`. This is not a term-former `Time_1 → Time_0` inside the object language. It is a metalevel operation on a derived-time closure that already stores the originating `t_0`. For general `Time_1` values, Lex uses a structured source-history map: `lift_0(t_0)` contributes a lift event, a derived closure contributes a derive event carrying the source time, pack id, witness digest, and rule digests, and `Toll(d,t_tol)` appends a toll event. The Qed-closed re-evaluation theorem preserves the source-time projection and records the new derivation pack; the full stable-fragment translation of arbitrary terms remains the open pack-evolution obligation below.

Define the metalevel projection `src_0(derive_1(t_0, P, w)) := t_0`.

**Proposition** (**re-evaluation soundness**)**.** For any `t_1 = derive_1(t_0, P, w_P)` and any `omega : Rewrite(P → P')` carrying replay transport, `reevaluate(t_1, omega) = derive_1(t_0, P', replay_witness(omega,w_P))`, and `src_0(reevaluate(t_1, omega)) = t_0`.

*Proof.* A derived legal time is the closure `(t_0, P, w_P)`: the frozen source fact, the pack under which the legal consequence was derived, and the pack-relative witness. A rewrite witness carries effective-date monotonicity, conservativity and type-preservation propositions, and replay transport. Re-evaluation along `omega : Rewrite(P → P')` applies witness replay to obtain `w_P'` and rebuilds the closure as `(t_0, P', w_P')`; the first projection is definitionally unchanged. The Rocq mechanization is `lex/formal/coq/Lex/PackReevaluation.v`: `re_evaluation_soundness`, `src_0_reevaluate`, `reevaluate_derivation_pack`, `reevaluate_id`, `reevaluate_compose`, `rewrite_effective_date_preserved`, `source_history_reevaluate_derived`, and `source_history_reevaluate_pack` close with `Qed`, and `Print Assumptions re_evaluation_soundness` reports closed under the global context.

On the `omega`-stable fragment, where every rule constant appearing in `G`, `e`, and `A` lies in `P.rules ∩ P'.rules`. The intersection is taken over rule digests, not over rule identifiers. Define the translation `tau_omega` by `tau_omega(r) = r` on preserved rule constants and extend it homomorphically to contexts, terms, and types. Write `G' = tau_omega(G)`, `e' = tau_omega(e)`, and `A' = tau_omega(A)`. The digest discipline is what makes "the same rule survives" a checkable side-condition: a rule whose digest is unchanged has verbatim-identical body, types, and priority metadata in both packs, so the homomorphic translation is literally the identity on preserved rules.

**Open obligation** (**pack-evolution soundness, stable fragment**)**.** If `G ⊢P e : A` and `omega : Rewrite(P → P')`, every rule constant used in the derivation of `G ⊢P e : A` has its digest in `P.rules ∩ P'.rules`, added rules in `P'` are inert for the translated derivation, and `omega` carries replay witnesses for every pack-relative derived-time closure used by the derivation, then `G' ⊢P' e' : A'`. If evaluating `e` under `P` and evaluating `e'` under `P'` both terminate in compliance verdicts, the two verdicts are equal.

*Proof target.* The expected proof is by induction on the typing derivation. The pack-sensitive leaves are rule constants and witnesses carried by `derive_1`. By hypothesis every such rule survives in `P'`; by the type-preservation obligation in `omega`, it has the same type in both packs, and by replay transport the derived-time witness can be rebuilt under `P'`. Every other constructor is translated homomorphically, so the ordinary induction for variables, lambda, application, let, match, and defeasible rules should go through unchanged. Verdict preservation requires the additional operational inertness argument that added rules in `P'` do not fire for this derivation.

**Worked example: Pakistan Companies Ordinance, 1984, and the 2010 fee rewrite.** Let `P_84` be the Sixth Schedule fee pack for the *Companies Ordinance, 1984*, and let `P_10` be the pack made effective on `2010-10-26` by S.R.O. 996(I)/2010. The baseline Sixth Schedule charged Rs. 200 for filing a document or return other than a charge instrument and Rs. 5,000 for recording a charge. The 2010 rewrite substituted Rs. 600 for an electronic non-charge filing, Rs. 1,500 for a physical non-charge filing, Rs. 5,000 for an electronic charge filing, and Rs. 7,500 for a physical charge filing.

The first case produces the same verdict after re-evaluation. Let `t_0^charge` be the frozen filing fact "a company filed a charge-registration document electronically on 2010-11-15 and paid Rs. 5,000." Under `P_84` the derived verdict `t_1^charge = derive_1(t_0^charge, w_P84)` is `Compliant`. Re-evaluating with `omega_84→10 : Rewrite(P_84 → P_10)` yields `reevaluate(t_1^charge, omega_84→10) = derive_1(t_0^charge, w_P10)` and the verdict remains `Compliant`, because the electronic charge fee is Rs. 5,000 in both packs.

The second case produces a different verdict. Let `t_0^return` be the frozen filing fact "a company filed an annual return electronically on 2010-11-15 and paid Rs. 200." A stale evaluator using `P_84` derives `t_1^return = derive_1(t_0^return, w_P84)` and reports `Compliant`. Re-evaluating under `omega_84→10` yields `derive_1(t_0^return, w_P10)` and changes the verdict to `NonCompliant`, because the 2010 pack raised the electronic non-charge filing fee from Rs. 200 to Rs. 600. The underlying filing fact is unchanged. Only the legal consequence derived from it changes.

**Remark (pack-version preorder).** Fix an authority and let `Packs_auth` denote the packs produced under it. Order them by `P ⩽_pack P'` iff there exists a well-typed witness `omega : Rewrite(P → P')`. Reflexivity is the identity rewrite; transitivity is rewrite composition (conservativity, type-preservation, effective-date monotonicity, and replay transport are each closed under composition). Under this ordering `Packs_auth` is filtered only for compatible conservative extensions: any two packs reachable from a common ancestor `P_0` admit a further pack `P_3` with `P_1, P_2 ⩽_pack P_3` when their surviving rule digests, type assignments, effective dates, and replay functions are mutually compatible. A rule's temporal validity window `[EffectiveDate(r, t0), EffectiveDate(Repeal(r), t_rep))` acts as a divisor on this filtration, a presence/absence marker on an up-set of pack versions. The operator `reevaluate` is the action of a pack-version inclusion on derivation witnesses, computable by a finite walk over the preserved-rule digest set plus the replay functions carried by `omega`. This framing is not used in any proof below; it records the structural picture that lines up the pack layer with the content-addressed proof bundles

of the companion Op paper and clarifies why `version : N` is a labelling, not a structural discriminant of pack identity.

**Conjecture (pack-evolution soundness for the modal and temporal subsystem).** The theorem above extends from the stable fragment to the full modal-temporal language: if `G ⊢P e : A` may contain the temporal modalities `@t A`, `diamond t A`, `box[t1,t2] A`, tribunal modals, and pack-indexed derived times, then every well-typed `omega : Rewrite(P → P')` induces translated judgments `G' ⊢P' e' : A'` that commute with `reevaluate` and preserve verdicts.

This is not proved here. The missing step is a single stratification argument showing that modal introduction and elimination commute with pack-indexed re-evaluation without creating either a hidden `Time_1 → Time_0` path or a cycle in the admissibility predicate. The same obstruction is what forces the companion Op paper's admissibility gate (§8.1) to reject the temporal coercions `Lift0` and `Derive1` from the Op compilation target: a derived time `derive_1(t_0, w_P)` carries the pack-relative witness `w_P` as part of its denotation, and Op has no operational-semantics-level primitive for transporting a pack-relative witness through a compiled trace until this conjecture is closed. The Lex admissible fragment (§5) and the Op admissibility gate therefore exclude temporal coercions for the same reason, not two unrelated reasons: neither layer can yet discharge the cross-pack commutation obligation this conjecture names.

### 3.4 Authority-Relative Interpretation

Compliance verdicts are indexed by the authority asserting them. In the core calculus, the **tribunal modal** `[T] A` is the type of `A`-evidence asserted under authority `T`; when `A : Prop`, it specializes to the proposition "A under authority T." Forming the modal type is a typing judgment on the calculus; inhabiting it requires the public authority-recognition judgment of Section 4.6, not a free term constructor in the core calculus. Different tribunals can form contradictory propositions about the same facts, and two tribunal-indexed terms cannot be silently identified. Converting between tribunals requires an explicit coercion:

```
coerce[ADGM_FSRA => Seychelles_FSA](verdict, bridge_witness)
```

The coercion requires a **bridge witness**, a term proving that the two authorities agree on the relevant interpretation. This witness may fail to exist, in which case the coercion is blocked. Divergence between authorities is represented as an explicit obstruction, not a runtime error.

**Authorities are unordered labels.** No hierarchy orders one tribunal above another. No operation aggregates a set of tribunal verdicts into a single network verdict. Legal authority is plural and sovereign; every aggregation scheme (weighted voting, seniority, FATF-member precedence) encodes a political judgment about which authority matters more. The calculus refuses that encoding.

The sole cross-tribunal mechanism is the bridge witness, which records that two authorities stipulate to the same verdict on the same facts. A bridge witness privileges neither side. When two tribunals disagree and no

bridge witness exists, the disagreement is type-theoretic content: a term typeable only under one tribunal does not coerce to the other, and a verifier working across both surfaces the divergence as an obstruction.

Multi-harbor composition, developed in the companion algebra paper, operates on the compliance tensors that tribunals produce (pointwise meet). Tribunal machinery sits at the level of verdict provenance. Cross-harbor aggregation lives one layer above, over verdicts, not over the authorities that produced them.

One fail-closed policy stands outside ordinary override: asserted sanctions non-compliance. In Lex, `sanctions-dominance(proof)` takes a proof of sanctions non-compliance and produces a hard block that cannot be overridden inside the ordinary tribunal, defeasibility, or mutual-recognition layer. This is a typing policy, not a universal claim that every sanctions statute admits no licenses, exemptions, delisting path, or shared-authority recognition. Those facts must enter before dominance fires, as local sanctions facts, a `SharedSanctionsAuthority` certificate, or a new rule-pack witness. The effect system enforces the boundary: the `sanctions_query` effect is distinguished, and the type checker tracks its presence.

### 3.5 Typed Discretion Holes

Lex distinguishes three states at any apparent hole site, rather than collapsing all uncertainty into one pending-authorization flag. Hart's core of settled meaning is mechanical. Hart's penumbra is discretionary: the law delegates judgment to an existing authority, and the answer has not yet been rendered. Dworkin's harder cases are unsettled: the present rule pack has no disposition, and no officer may answer without changing the law itself. The three states:

1. **Mechanical.** The question lies within the core of settled meaning. The evaluator computes the answer directly, and the audit trail records that no human judgment was needed.
2. **Discretion.** The question lies in Hart's penumbra. A legally recognized authority exists, but the answer has not yet been rendered. The derivation suspends pending a proof-carrying authorization witness.
3. **Unsettled.** The law has no present disposition. No authorized individual may supply an answer under the current rule pack. The derivation suspends pending legislative or regulatory change.

In the core calculus:

```
MechanicalHole(has_natural_person_director)                 : Bool
DiscretionHole(ADGM.FSRA, fit_and_proper)                   : ComplianceVerdict
  scope { jurisdiction: ADGM, entity_class: AuthorizedFirm }
UnsettledHole(DigitalAssets, tokenized_beneficial_ownership): ComplianceVerdict
```

The first shape is degenerate but important: a `MechanicalHole` marks a site that was once legally interesting but is now within the rule pack's settled core, so the type checker discharges it automatically. A `DiscretionHole(auth, h)` marks a site where existing law delegates judgment to `auth`. An

`UnsettledHole(domain)` marks a stronger failure: no lawful answer exists until the rule pack itself changes. It cannot be filled by an officer, board member, or regulator acting under the existing pack.

Only the discretionary case uses the existing fill path:

```
fill(fit_and_proper, Compliant, w)    where  w : PCAuth(auth, h, Compliant, request(h,Complia
```

**PCAuth** (**Proof-Carrying Authorization**) is the quorum-indexed witness family binding the fill to its whole admission context: (1) the identity of each signer who supplied the judgment, (2) the authority under which each signer acts, (3) the scope in which the authority applies, (4) the exact hole and value, and (5) the request, pack, and proof-context digests against which the fill was solicited. The single-signer record below is the `k = 1` projection used for exposition; the quorum form is the canonical verifier target.

```
FillMode(h) = Creative | Applied(PrecedentRef(h))

PCAuth(auth, h, v, r) = {
  signer        : Did,                                                  , decentralized ide
  role          : AuthorityRole(auth),                                  , proof that signer
  scope_ok      : ScopeWitness(h.scope, v),                             , proof that v lies
  justification : Option(Text),                                         , optional public-f
  mode          : FillMode(h),                                          , novel reasoning o
  ledger_ref    : LedgerRef,                                            , append-only justi
  request_hash  : FillRequestHash,                                      , digest of the eva
  pack_digest   : PackDigest,                                           , governing rule-pa
  context_digest: ContextDigest,                                        , surrounding proof
  timestamp     : Time_0,                                               , when the judgment
  signature     : Ed25519Sig(PCAuthPayload(
                    signer, auth, h, v, role.chain_digest, h.scope,
                    timestamp, request_hash, mode,
                    digest(justification), ledger_ref,
                    pack_digest, context_digest))
}
```

Scope is a meet-semilattice element, not a record shape. Fix once for the rest of the paper: a `Scope` for a hole `h` of value type `tau_h` is a decidable downward-closed predicate `S : Pi x : tau_h. Prop`; the semilattice operation `S cap S'` is pointwise conjunction (`(S cap S')(x) := S(x) /\ S'(x)`), the top element is the constantly-true predicate, and inclusion `S subseteq S'` means pointwise implication. A `ScopeWitness(S, v)` is a proof term of `S(v)`. Concrete presentations such as `{ jurisdiction: ADGM, entity_class: AuthorizedFirm }` (§7.5) are named atomic predicates joined by `cap`; they are surface sugar for elements of this lattice. The hole's declared scope `h.scope` is such an element; the chain-intersected scope of §4.8 is another element; "the chain scope contains `h.scope`" means the chain

predicate entails `h.scope`. Throughout, scope is neither a type nor a record, and the lattice structure is what the paper relies on.

The `role` field is the critical trust anchor. The role is witnessed by the authority named in the hole; it is never self-asserted. The intended resolution is an identity primitive: a verifiable credential attesting that a specific person holds a specific role (e.g., "FSRA-authorized compliance officer") at the authority named by `auth`. The optional `justification` field is audit-relevant text explaining the reasoning. Justifications are published to an append-only ledger, addressed by `ledger_ref`, so later reviewers can inspect why a fill was made. The `mode` field distinguishes `Creative` fills, where the authority advances novel reasoning, from `Applied` fills, which cite an existing precedent reference.

**Theorem (finite discretion-hole event-union core).** Under a finite observation space and a constructed reduction mapping every hole-forgery observation either to an EUF-CMA signature-forgery observation or to a credential-chain overhead observation, the number of successful hole-forgery observations is bounded by the sum of the two target counts. The Rocq target `finite_observation_event_union_bound` in `PaperMechanization.v` is Qed-closed. Instantiating the abstract event predicates with Ed25519 EUF-CMA games and a concrete credential-issuance verifier remains the cryptographic binding step, not a missing proof of the counting lemma.

The companion paper *Op: A Typed Bytecode for Compliance-Carrying Operations* fixes the target proof-bundle transport form of this witness in its Section 8.3, "PCAuth Verifier and Transport." The current public mechanized fill case treats the witness as an uninterpreted authority/digest/timestamp payload transported through compilation; delegated revocation, expiration, multi-signer quorum transport, bulk verification, and cross-zone bridge validation remain target verifier obligations unless separately cited as closed.

Revocation is captured by a separate typed artifact. The type `Revoke(c) : RevokedCred` denotes a revocation certificate for a credential `c : Credential`, itself signed by the issuer of `c` and bulletin-stamped at a public time `t_r`. A verdict derived from a filling `fill(h, v, w)` is tagged `Tainted` if the issuing credential is revoked at a bulletin time `t_r > t_w` (the filler was authorized when they filled, but subsequent revocation casts the verdict into a tainted state) and `Invalid` if `t_r ⩽ t_w` (the credential was already revoked at the moment of filling, so the witness should never have been accepted). The temporal ordering is supplied by the public bulletin, not by the filler (Section 4.9).

*Proof sketch.* A forged witness either (a) contains a forged `Ed25519Sig` binding signer, authority, hole, value, delegation-chain digest, scope digest, timestamp anchor, fill request, fill mode, justification digest, ledger reference, pack digest, and context digest, which reduces to an EUF-CMA forgery, or (b) contains a legitimate signature from a party whose `AuthorityRole(auth)` credential was issued outside the sanctioned issuance procedure, which contradicts the credential-issuance soundness assumption. The failure probability is the union of the two.

The intended security reduction has two institutional assumptions: the cryptographic soundness of the signature scheme and the administrative soundness of credential issuance. Both are standard targets for

attack in any PKI-style system. Open problems below this level, revocation of PCAuth witnesses when a filer's credential is later rescinded, root-of-trust establishment for a multi-authority corridor, are enumerated in Section 13. The obligation authenticates the author and integrity of the fill metadata; it does not certify the legal correctness of the justification itself.

**PCAuth** (**Proof-Carrying Authorization**) is a cryptographic attestation binding the authority, the specific discretion hole, the exact value supplied for that hole, and the fill-request context that made the value admissible. If hole `h` has type `tau_h`, `v : tau_h`, and `r : FillRequest(auth,h,tau_h,pack_digest,context_digest)`, then `PCAuth(auth, h, v, r)` is the witness type:

```
PCAuth(auth, h, v, r) = {
  quorum       : Nat,                              -- required signer threshold
  signers      : Vec n Did,                        -- n candidate signers
  depth        : Vec n Nat,                        -- realized delegation depths
  authority    : Vec n AuthorityChain(auth, signers[i], depth[i]),
  scope_ok     : ScopeWitness(h.scope, v),         -- proof that v lies in the authorized sco
  timestamp    : Vec n Time_0,                     -- when each judgment was made
  anchor       : Vec n LinkedTimestamp(payload_hash[i]),
  request_hash : FillRequestHash,                  -- digest of r
  mode         : FillMode(h),
  ledger_ref   : LedgerRef,
  pack_digest  : PackDigest,
  context_digest : ContextDigest,
  payloads     : Vec n PCAuthPayload(signers[i], auth, h, v,
                     authority[i].chain_digest, h.scope,
                     timestamp[i], anchor[i],
                     request_hash, mode, ledger_ref,
                     pack_digest, context_digest),
  signatures   : Vec n Ed25519Sig(payloads[i]),
  distinct     : DistinctSigners(signers),
  quorum_ok    : quorum ⩽ n
}
```

We write `PCAuth_k(auth, h, v, r)` for the quorum-refined form of the same witness family, namely a witness of type `PCAuth(auth, h, v, r)` whose `quorum` field is exactly `k`. The `authority` field records a root authority credential together with a bounded delegation chain terminating at each signer. The `anchor` field is a linked timestamp in the sense of Haber and Stornetta (1991), so each signed payload is cryptographically committed to a public log position. No free `val` parameter remains: the value is part of the witness type itself, and every signature is over the exact canonical payload, including authority, hole,

value, fill request, pack digest, and proof-context digest. `DistinctSigners` prevents duplicate signer counting; replay into another hole, request, pack, or proof context is blocked only because those digests are signed.

Indexing, stated once. `PCAuth` is a family in the Lex category with families of §9: writing `Gamma_pc := (auth : Authority, h : HoleId, v : tau_h, r : FillRequest(auth,h,tau_h))`, the family is `PCAuth in Ty(Gamma_pc)`, and its display map projects the total space `Sigma w : PCAuth(auth, h, v, r)` onto the base `Gamma_pc`. The base is itself a dependent context: `tau_h` depends on `h : HoleId` via the pack-global map `h ⟼ tau_h` supplied by the hole signature of §4.7, and `r` commits the current pack and proof context. A `fill(h, e, w)` is a section of this display map at the indexing point `(auth, h, e, r)`, and `VerifyPCAuth` is the decision procedure that re-checks that section's defining data. The CwF structure of §9 is the index category for `PCAuth`; the presheaf model of §10 interprets the family as the partial section whose support is exactly the set of authority-contexts at which a witness is recorded.

We write `VerifyPCAuth(W, h, v, r, check)` for the verifier judgment of signature `PCAuth(auth, h, v, r) x HoleId x tau_h x FillRequest x AdmissionCheck → Bool`; it re-checks signatures on the canonical payload, binds the witness threshold to the hole policy `required(h)`, checks signer membership in `committee(h)` and the deployment `n_max` bound, linked timestamp anchors, authority-chain validity, non-revocation of every credential in the authority chain at the admission/fill check snapshot, and the delegation depth bound. Later revocation does not rewrite the historical fill; it tags downstream use as `Tainted` or `Invalid` according to §4.9.

**Theorem (quorum acceptance unfolding).** Let `W : PCAuth_k(auth, h, v, r)` and let `VerifyPCAuth(W, h, v, r, check)` accept against the hole policy. Then there exists a finite list `Q` of at least `required(h)` signer attestations whose signer identities are distinct, whose signers lie in `committee(h)`, and such that, for every `a in Q`, the proof bundle contains a valid Ed25519 signature on the canonical payload binding `(protocol_tag, canon_version, hash_suite, sig_alg, a.signer, auth, h, v, request_hash(r), pack_digest(r), context_digest(r))`, a linked timestamp anchor for that payload, an authority chain rooted at `auth`, ending at `a.signer`, scoped to `h`, whose realized length does not exceed the recorded and policy depth bounds, and no credential in that chain was revoked at or before the admission snapshot. Consequently, an accepted filled-hole record under this admission request exposes at least `required(h)` distinct valid signer attestations over the exact policy-bound payload the verifier checked.

*Proof.* The verifier predicate is a finite conjunction: `quorum ⩽ length attestations`, `NoDup(map signer attestations)`, and pointwise validity of every attestation at `t_check`. Pointwise validity expands to exact payload binding (`protocol_tag`, `canon_version`, `hash_suite`, `sig_alg`, `authority = auth`, `hole = h`, `value = v`, `request = request_hash(r)`, `pack = pack_digest(r)`, `context = context_digest(r)`), signer membership in the policy committee, delegation depth within the recorded and policy bounds, signature valid-

ity, timestamp-anchor validity, authority-chain validity, and non-revocation at the admission check time. Taking `Q` to be the accepted attestation list gives the result. The Rocq mechanization is `lex/formal/coq/Lex/PCAuthQuorum.v`: `quorum_acceptance_unfolding`, `accepted_filled_hole_quorum`, `quorum_exact_payload`, `quorum_depth_bound`, `quorum_signature_payload_binding`, and `quorum_anchor_chain_not_revoked` close with `Qed`, and `Print Assumptions quorum_acceptance_unfolding` reports closed under the global context. This theorem is structural verifier extraction over an honest verifier record; it is not a cryptographic unforgeability theorem or a proof that arbitrary receipt generators are sound.

**Open obligation (PCAuth forgery reduction).** Fix a deployment in which the maximum quorum width is bounded above by `n_max` (a deployment-time configuration parameter set in the authority pack; its existence is assumed throughout). Fix the verifier for `PCAuth_k(auth, h, v, r)` for any `k ⩽ n_max`, and assume Ed25519 is EUF-CMA secure with insecurity function `epsilon_{Ed25519}(t)` against time-`t` adversaries in the standard model (Bellare-Rogaway 1996; for Ed25519 specifically Bernstein-Duif-Lange-Schwabe-Yang 2012, with EUF-CMA reduction to the strong-CMA variant of Schnorr signatures). Let `A` be a probabilistic polynomial-time adversary running in time `t_A` that, on input the public verification keys of `n_max` honest signers, outputs an accepted PCAuth witness for a fresh canonical payload containing at least one signature on that payload that was not previously returned by the signing oracle for the relevant signer. Let `epsilon_A(t_A)` denote `A`'s success probability. The target statement is that there exists a probabilistic polynomial-time adversary `B` against the EUF-CMA security of Ed25519, running in time `t_B = t_A + O(n_max · q_{sig})` where `q_{sig}` is the maximum number of signing queries `A` issues, such that

```
epsilon_{Ed25519}(t_B) ⩾ epsilon_A(t_A) / (k · n_max).
```

The reduction boundary is signer honesty, not legal omniscience. If `c` committee credentials are corrupted, the cryptographic statement applies only to accepted witnesses whose canonical quorum subset contains at least one honest signer signature that was not obtained from that signer's signing oracle; if `c ⩾ k`, a corrupted quorum can sign a payload without violating EUF-CMA. Likewise, if an authorized honest signer intentionally signs a legally wrong but policy-admissible payload, the signature is not a forgery. That event belongs to the credential-policy, disciplinary, or appeal layer. A complete threshold theorem therefore assumes `c < k`, honest signers sign only payloads satisfying their local authorization predicate, and the verifier's committee and revocation snapshots are the ones fixed by the authority pack.

The factor `k · n_max` decomposes as: `n_max` for slot guessing across the worst-case maximum quorum width, multiplied by `k` for the cost of forging a `k`-of-`n` quorum (the adversary must commit to which of the `k` signature slots in the accepted witness it claims as the fresh forgery, since acceptance requires `k` valid signatures and any of them might be the new one).

*Proof target.* The reduction proceeds in two stages: a slot-guessing stage (factor `n_max`) and a quorum-position stage bounded by the length of the accepted canonical quorum subset. The public proof obligation

must first fix a canonical k-attestation subset extracted from the accepted list, because an accepted witness may contain more than k valid attestations. The hybrid argument is then over the n_max honest signer slots and that canonical subset.

(1) *Slot guessing.* The reduction B receives the EUF-CMA challenge public key pk_* and the adversary's claim about the universe of n_max honest signer identities. B samples a guess i_* in {1, ..., n_max} uniformly at random and embeds pk_* in slot i_* of the public-key vector. For every other slot j ≠ i_*, B samples a fresh Ed25519 keypair and registers pk_j honestly, retaining sk_j to answer signing queries for s_j.

(2) *Hybrid simulation of signing queries.* For any signing query A issues for slot i_* on a canonical PCAuth payload m for signer s_{i_*}, B forwards m to the EUF-CMA challenger and returns the response. For any signing query for a slot j ≠ i_*, B answers using the locally retained sk_j. This perfectly simulates A's view of the signing oracle for all slots.

(3) *Quorum-position guessing.* When A outputs an accepted witness W, the verifier extracts the canonical quorum subset Q_k of k attestations from the accepted attestation list. B samples a position p_* in {1, ..., k} uniformly at random and inspects the p_*-th signature in Q_k. By verifier acceptance (the quorum acceptance theorem above), this is a valid Ed25519 signature on the exact canonical payload for some signer identity s_{p_*}.

(4) *Forgery extraction.* The reduction succeeds when (a) s_{p_*} = s_{i_*} (the inspected position points to the slot in which B embedded the EUF-CMA challenge key), and (b) the canonical payload for s_{i_*} was never queried to the signing oracle (the fresh-signature condition guaranteed by A's freshness premise). When both hold, the signature extracted from position p_* is an EUF-CMA forgery against pk_* on that payload. Condition (a) holds with probability 1 / (k · n_max) by independent uniform sampling of p_* and i_*. Condition (b) holds whenever A succeeds and the canonical subset contains the fresh attestation; formalizing this selection invariant is part of the open reduction target.

(5) *Hybrid composition.* The hybrid argument over the n_max honest signer keys is the standard one (Bellare-Rogaway 1996, §3): the adversary's view in slot i_* is identically distributed under either the EUF-CMA challenger's response or B's local signing, so the embedding does not change A's success probability conditional on the slot-guess being correct. The factor n_max is therefore tight in the standard slot-guessing reduction; replacing the slot-guess by partitioning the public-key space (Cramer-Damgaard 1996) would tighten the bound asymptotically but is left as a deployment-time optimisation outside this obligation.

The runtime overhead O(n_max · q_{sig}) is the cost of locally answering signing queries for the n_max - 1 non-challenge slots and of constructing the n_max-slot public-key vector. This reduction is not mechanized in the current Rocq development; it is a cryptographic obligation whose adversary model,

canonical-quorum extraction, credential-issuance assumptions, and bulletin assumptions must be fixed before it can be counted as a theorem.

If discharged, the bound `epsilon_A(t_A) ⩽ (k · n_max) · epsilon_{Ed25519}(t_B)` is the headline statement: a PCAuth forgery against a `k`-of-`n_max` quorum costs at most a `k · n_max` factor over a single Ed25519 forgery. For a typical deployment with `n_max = 7` and `k = 4`, the loss factor is 28. The target reduction is in the standard model; the random-oracle model would tighten the bound by removing the slot-guessing step but would weaken the assumption side of the comparison.

Credential issuance, revocation, and delegation are therefore typed institutional objects rather than ambient side conditions. Their typing rules appear in Section 4.7, and Op re-checks them independently at dispatch.

Filled discretionary judgments can be lifted into first-class precedent values:

```
Precedent(fit_and_proper, past_fills) : PrecedentChain(fit_and_proper)
```

This is consultative rather than binding. Following Levi's *An Introduction to Legal Reasoning* (1949), precedent is reasoning by example: later fillers may consult the chain for guidance, cite it in an `Applied` fill, or depart from it with a `Creative` fill that begins a new branch. Lex types that distinction rather than pretending all fills are alike.

Filled discretionary judgments can also be appealed:

```
Appeal(fill_2025_01, ADGM.AppealsPanel)
  : AppealedFill(fit_and_proper, fill_2025_01, ADGM.AppealsPanel)
```

An appeal is a higher-authority re-fill of the same hole. The original fill remains in the record; downstream verdicts derived from it are tagged `Appealed` until they are re-derived under the higher-authority disposition. Revocation is typed supersession, not erasure.

The derivation trace can now distinguish three outcomes: fully mechanical evaluation, discretionary evaluation with an attributed judgment and public justification, and genuine unsettled law requiring pack evolution. The boundary is typed. The audit trail is complete.

---

## 4. The Core Calculus

The core calculus is a standard dependent type theory with five extensions: defeasible rules (Section 4.4), temporal sorts (Section 4.5), tribunal modals (Section 4.6), effect rows (Section 4.3), and typed discretion holes (Section 4.7). We present the standard foundation briefly and the extensions in detail. The calculus is the contribution. The admissible fragment (Section 5) is the implemented subset. The gap between them is the research agenda.

### 4.1 Universe Hierarchy

Lex has a cumulative universe hierarchy (each universe level includes all types from lower levels) with three sort families:

- **Type_l** for l = 0, 1, 2, ..., the standard universe hierarchy for computational types.
- **Prop**, the proof-irrelevant sort, living at Type_1. Propositions in Prop are erased at runtime; only their provability matters.
- **Rule_l**, a documentation sort for compliance rules, stratified identically to `Type_l`, with no additional typing discipline. The distinction is a semantic marker, normative rather than descriptive, that appears in diagnostic output and has no formal consequence in the typing rules.

Additionally, **Time_0** and **Time_1** are temporal sorts at universe level 0.

The typing rules for sorts:

```
------------------- (Sort-Type)
 G |- Type_l : Type_{l+1}

------------------- (Sort-Prop)
 G |- Prop : Type_1

------------------- (Sort-Rule)
 G |- Rule_l : Type_{l+1}

------------------- (Sort-Time)
 G |- Time_i : Type_0     (i in {0, 1})
```

In plain terms: **Sort-Type** says that each universe of types lives one level above itself, the type of all small types is itself a larger type, preventing self-reference paradoxes. **Sort-Prop** says that propositions (statements that are either provable or not) live inside `Type_1`. **Sort-Rule** is a documentation sort: compliance rules have the same hierarchical structure as types, and the marker is preserved through elaboration and diagnostic output but imposes no additional typing constraint. **Sort-Time** says that both temporal sorts, frozen historical time and derived legal time, are ground-level types.

Universe levels are expressions built from natural number literals, level variables, successor (`l + n`), and maximum (`max(l1, l2)`). The metatheoretic universe hierarchy is countably infinite (indexed by ⊠); the implementation imposes a finite bound on absolute level values for efficient level resolution, and this bound does not affect the metatheory.

**Prop Sort** Lex follows ordinary Coq-style `Prop` rather than a judgmentally proof-irrelevant sort: proof irrelevance is *propositional*. It is not definitional. Following Martin-Löf (1984), Werner (1997), and Pfenning

(2001), proof terms in `Prop` participate in typing and in the identity type; runtime evaluation erases them. We write `bottom_prop : Prop` for propositional falsity, `not P` for `P → bottom_prop`, and `P or Q` for proof-irrelevant disjunction.

```
G |- A : S    S in {Prop, Type_i}    G |- x : A    G |- y : A
------------------------------------------------------------- (Id-Form)
                 G |- Id(A, x, y) : Prop

G |- x : A
-------------------------- (Id-Intro)
G |- refl(x) : Id(A, x, x)

G, x:A, y:A, p:Id(A, x, y) |- C(x, y, p) : S    S in {Prop, Type_i}
G, z:A |- d(z) : C(z, z, refl(z))
G |- a : A    G |- b : A    G |- q : Id(A, a, b)
------------------------------------------------ (J)
             G |- J_{C,d}(q) : C(a, b, q)
```

**Id-Form** introduces propositional equality for both computational types and propositions. **Id-Intro** is reflexivity. **J** is the ordinary Martin-Löf eliminator: to prove a property of every equality proof, it suffices to prove the reflexive case. Lex uses J intensionally; no univalence axiom or higher-path structure is assumed, and no claim of HoTT-style identity beyond J is made. In the current classical fragment, transport arguments using J are discharged classically rather than by a separate constructive normalization proof.

```
G |- P : Prop    G |- p : P    G |- q : P
---------------------------------------- (Prop-Irrelevance)
        G |- proofirr(p, q) : Id(P, p, q)

G |- P : Prop    G |- d : Dec(P)
------------------------------- (Dec-LEM)
      G |- decide(d) : P or not P

G |- P : Prop
-------------------------------- (Classic-LEM)
      G |- classic(P) : P or not P
```

**Prop-Irrelevance** is the proof-irrelevance commitment: for any two proofs of a proposition, Lex can construct a proof that they are equal. This is *not* a conversion rule. Two proofs may remain judgmentally distinct even when `proofirr(p, q)` inhabits `Id(P, p, q)`; the equation is propositional rather than definitional. This matches ordinary Coq `Prop` rather than `SProp`. **Dec-LEM** provides excluded middle constructively whenever a decision procedure is available: equality on finite prelude types and Ed25519

signature verification are representative examples. **Classic-LEM** is the full classical axiom schema for compliance propositions, matching Coq's `Classical_Prop.classic` when the rule author opts into classical reasoning.

**Consistency of the Prop-sort design (open).** The combination of Id-Form, J, Prop-Irrelevance, Dec-LEM, Classic-LEM, Or-Elim-Prop, Subset-Form/Intro/Elim, and Squash is *not* proved consistent in this paper. The justification "stable under classical reasoning because `classic(P)` itself inhabits `Prop`" is exactly that, a justification, not a proof. The combination is closely modelled on Coq's `Prop` plus its `Classical_Prop.classic` axiom, which Werner (1997, "Sets in types, types in sets") proves consistent relative to ZFC together with one strongly inaccessible cardinal per universe level. Our fragment additionally restricts `Prop` elimination into computational sorts to singleton propositions, which is a strictly stronger discipline than Coq stdlib's; it does not introduce new consistency obligations beyond what Coq's `Prop` + `Classical_Prop` already discharge under Werner's argument. A full standalone consistency proof for the exact rule set above (including the interaction of `Subset` with `Squash` and with the dependent eliminator J on `Prop`-sorted motives) is open and listed in §13.4.

Elimination from a proof in `Prop` into a computational type is restricted to *singleton propositions*: propositions whose eliminators reveal no runtime branch information. The class includes empty and unit propositions and proof-irrelevant identity proofs. It excludes ordinary disjunction, truncated existence, and subset-proof projections. In particular, there is no rule that case-analyzes a proof of `P or Q` to synthesize a value in `Type_i`.

```
G |- u : P or Q    G, p:P |- c_P : C    G, q:Q |- c_Q : C    G |- C : Prop
------------------------------------------------------------------------------ (Or-Elim-Prop)
                      G |- orElim(u, c_P, c_Q) : C

G |- A : Type_i    G, x:A |- P(x) : Prop
---------------------------------------- (Subset-Form)
           G |- {x : A | P(x)} : Type_i

G |- a : A    G |- p : P(a)
------------------------------------------ (Subset-Intro)
         G |- <a, p> : {x : A | P(x)}

G |- s : {x : A | P(x)}
----------------------- (Subset-Elim-1)
        G |- pi_1(s) : A

G |- s : {x : A | P(x)}
---------------------------------- (Subset-Elim-2)
```

```
    G |- pi_2(s) : P(pi_1(s))
```

Subset types package a computational witness together with an erased compliance proof. They are the proof-irrelevant refinement layer for Lex: a term of type `{x : A | P(x)}` carries a runtime value of type `A` and a proof that the value satisfies `P`. This is the source-language analogue of the refinement discipline used by Op for compliance-carrying artifacts: the carrier survives execution, while the proof certifies the refinement and erases.

```
G |- A : Type_i
------------------------- (Squash-Form)
        G |- ||A|| : Prop

G |- a : A
------------------------- (Squash-Intro)
       G |- |a| : ||A||

G |- u : ||A||    G |- C : Prop    G, x:A |- c : C
------------------------------------------------- (Squash-Elim)
           G |- squashElim(u, x.c) : C
```

The squash type `||A||` forgets computational content while retaining mere inhabitation. It is useful for existence claims where the witness does not matter operationally; we write `exists x : A. P(x)` as shorthand for `||{x : A | P(x)}||` when the witness is irrelevant. Like disjunction, squash eliminates only into `Prop`.

Let `erase(-)` delete every subterm whose type is in `Prop`, project `<a, p>` to `erase(a)`, and drop proof-only arguments from eliminators. Let `Lex_erased` be the sub-calculus obtained by removing the `Prop` sort and all constructors whose only outputs lie in `Prop`.

**Theorem** (**Prop erasure**). If `G ⊢ e : A [rho]`, `G ⊢ A : Type_l`, and every erased Prop-sorted subderivation has empty computational effect row, then `erase(G) ⊢_{Lex_erased} erase(e) : erase(A) [rho]`.

*Proof sketch.* By induction on the typing derivation. Computational forms preserve their typing under erasure; proof constructors disappear; subset values erase to their first projection; and squash and disjunction eliminators remain inside `Prop` and are therefore removed before runtime evaluation. The empty-effect premise is load-bearing: if a Prop subderivation performs an oracle query or carries a visible effect row, erasing its term while preserving the outer row would leave an effect with no erased computation. A stronger variant would preserve such evidence in an erased certificate layer instead of requiring emptiness; that variant is part of the open proof-erasure program in §13.4. ■

### 4.2 Terms

The core term language, using standard dependent type theory notation with Lex-specific extensions. Variables use de Bruijn indices (a nameless variable representation where each variable is identified by its binding distance rather than a name), written as subscript naturals when disambiguation is needed.

Typing judgments carry an effect row. We write `G ⊢ e : A [rho]` for "under context `G`, expression `e` has type `A` and requires the effects `rho`." Effect rows are discussed in Section 4.3; the introduction and elimination rules below thread them through every binding form.

**Lambda abstraction and dependent function type (Pi):**

```
G |- A : Type_i [empty]    G, x:A |- B : Type_j [empty]
------------------------------------------------------ (Pi-Form)
    G |- Pi(x : A) [rho]. B : Type_{max(i,j)} [empty]

G, x:A |- b : B [rho]
---------------------------------------- (Lam-Intro)
 G |- lam(x : A). b : Pi(x : A) [rho]. B [empty]

G |- f : Pi(x : A) [rho_1]. B [rho_2]    G |- a : A [rho_3]
------------------------------------------------------------ (App)
        G |- f a : B[a/x] [rho_1 U rho_2 U rho_3]
```

**Pi-Form** says: if `A` is a type and `B` is a type (possibly depending on a value of type `A`), then the dependent function type "given an `A`, produce a `B` while incurring effects `rho`" is itself a type. The output type `B` can mention the input value `x`, which lets a compliance rule's return type vary by jurisdiction. **Lam-Intro** says: a lambda whose body requires effects `rho` is introduced as a function whose Pi carries the same `rho`; constructing the function itself is pure. **App** says: applying a function incurs the union of the Pi's latent effects, the function-position's effects, and the argument's effects, no effect is silently dropped.

**Let binding:**

```
G |- A : Type_i [empty]    G |- e : A [rho_1]    G, x:A |- b : B [rho_2]
------------------------------------------------------------------------ (Let)
     G |- let x : A := e in b : B[e/x] [rho_1 U rho_2]
```

**Let** says: the effects of a let-binding are the union of the initializer's effects and the body's effects. Naming a value neither erases nor hides its effect row.

**Dependent pair type (Sigma) and projections:**

```
G |- A : Type_i [empty]    G, x:A |- B : Type_j [empty]
------------------------------------------------------ (Sigma-Form)
```

```
G |- Sigma(x : A). B : Type_{max(i,j)} [empty]
```

**Sigma-Form** says: a dependent pair bundles a value of type `A` together with a value of type `B` that may depend on the first value. Where a Pi type is a function ("given an `A`, produce a `B`"), a Sigma type is a pair ("an `A` together with a `B` that may depend on which `A`"). In compliance terms, this could package a jurisdiction together with jurisdiction-specific evidence.

**Quantification over finite legal collections:**

The `Pi`-form above is the general dependent function space of type theory. It is not the surface form for legal quantification over a company's finite rosters of directors, shareholders, officers, or beneficial owners. Lex therefore adds separate bounded quantifiers ranging over explicit finite collections.

A collection `Collection(A)` carries a finite enumeration `enum(C) = [c_1, ..., c_n]`; concrete instances include lists, sets, and multisets. The bounded quantifiers range only over such collections:

```
G |- C : Collection(A) [rho_C]    G, x:A |- P(x) : Prop [rho_P]
------------------------------------------------------------------------ (Forall-Bounded)
G |- forall x : A in C. P(x) : Prop [rho_C U rho_P]

G |- C : Collection(A) [rho_C]    G, x:A |- P(x) : Prop [rho_P]
------------------------------------------------------------------------ (Exists-Bounded)
G |- exists x : A in C. P(x) : Prop [rho_C U rho_P]

G |- C : Collection(A) [rho_C]    G, x:A |- P(x) : Prop [rho_P]
------------------------------------------------------------------------ (ExistsUnique-Bounde
G |- exists! x : A in C. P(x) : Prop [rho_C U rho_P]
```

Operationally, if `enum(C) = [c_1, ..., c_n]`, then `forall x : A in C. P(x)` reduces to `P(c_1) ∧ ... ∧ P(c_n)` and `exists x : A in C. P(x)` reduces to `P(c_1) ∨ ... ∨ P(c_n)`. The empty conjunction is `top`; the empty disjunction is `bottom`. Unique existence expands to existence plus a uniqueness side condition:

`exists! x : A in C. P(x)` reduces to `(exists x : A in C. P(x)) ∧ ∧_{i,j} ((P(c_i) ∧ P(c_j)) → c_i = c_j)`.

This is the quantificational shape that statutes use when they say "every director shall ...", "some shareholder may ...", or "the company shall have exactly one secretary."

Bounded existential quantification also supports witness extraction:

```
G |- h : exists x : A in C. P(x) [rho]
------------------------------------------------------- (Choose)
G |- choose(h) : Witness(A, C, P) [rho]
```

Here `Witness(A, C, P)` is a prelude certificate type with constructor `Found(a, m, p)` where `a : A`, `m : Member(a, C)`, and `p : P(a)`. Because `choose` scans the finite enumeration of `C` and stops at the first witness, extraction is bounded by `|C|` and does not require the readmission of general Sigma projection to the admissible fragment. If `h` proves unique existence, the witness returned by `choose(h)` is canonical up to definitional equality on `A`.

Some legal rules quantify over rules rather than over entity data. For that purpose the full calculus reserves a higher-order meta-quantifier:

```
Pi_meta : (Rule_l → Prop) → Prop
```

It expresses meta-rules such as "every rule granting privilege X must also impose constraint Y." This constructor is outside the admissible fragment: its domain is the open-ended rule space rather than an explicit finite collection, so checking it requires reflection over rule syntax and metadata rather than bounded iteration over enumerated data.

**Proposition (Finite-collection quantifiers are admissible).** Let `C` be a closed prelude collection with finite enumeration `[c_1, ..., c_n]`. If every instantiated predicate `P(c_i)` is admissible, then `forall x : A in C. P(x)`, `exists x : A in C. P(x)`, `exists! x : A in C. P(x)`, and `choose` over a proof of bounded existence are admissible.

*Proof sketch.* The bounded universal expands to a finite conjunction; the bounded existential expands to a finite disjunction; unique existence adds only a finite family of equality checks over pairs drawn from the same finite enumeration; and `choose` performs a linear scan over that enumeration until it finds a witness. Each step is bounded by the cardinality of `C`, so the reduction adds finite search but no unbounded recursion or open-ended proof search. The argument is the standard bounded-quantifier move in decision procedures over finite search spaces; see Bradley and Manna (2007). ■

**Theorem (Bounded Skolemization).** Let `C : Collection(A)` and `D : Collection(B)` be finite. If `forall x : A in C. exists y : B in D. P(x, y)` is inhabited, then there exists a Skolem function `f : A → B` such that `forall x : A in C. P(x, f(x))` and `f(x) in D` for every `x in C`.

*Proof.* Write `enum(C) = [c_1, ..., c_n]`. An inhabitant of the premise yields, for each `c_i`, a proof of `exists y : B in D. P(c_i, y)`. Applying `choose` to each such proof returns a witness `d_i in D` together with a proof of `P(c_i, d_i)`. Define the Skolem function on the quantified domain by `f(c_i) = d_i`. Then `P(c_i, f(c_i))` holds for each enumerated element, so `forall x : A in C. P(x, f(x))` follows by conjunction introduction over the finite expansion of the bounded universal. This is the bounded finite-domain counterpart of Skolem's classical transform from existential witnesses to Skolem functions (Skolem 1920). ■

**Worked examples.** The Seychelles International Business Companies Act 2016, s.66, "A company shall have at least one director who is a natural person," is a bounded existential over the directors collection:

```
exists d : Director in c.directors. NaturalPerson(d)
```

Threshold statutes are the same finite-collection discipline expressed through cardinality. The Companies Act 2006, s.154 requirement that a public company have at least two directors is:

```
|c.directors| ⩾ 2
```

and, when a jurisdiction counts only directors satisfying a predicate, the same operator appears as a filtered cardinality:

```
|filter NaturalPerson c.directors| ⩾ 2
```

**Pattern matching (dependent):**

```
G |- e : T [rho_0]    G, x:T |- P : S [empty]    S in {Prop, Type_k}
if G |- T : Prop and S = Type_k, then SingletonProp(T)
(for each branch C_i xs => b_i, where C_i : Pi(xs : Args_i). T:
    G, xs : Args_i |- b_i : P[C_i(xs)/x] [rho_i])
------------------------------------------------------------- (Match-Dep)
    G |- match e return x. P with | C_1 xs_1 => b_1 | ... : P[e/x] [rho_0 U U_i rho_i]
```

**Match-Dep** says: to inspect a value `e` by cases, you supply a *motive* `x.P` that states how the return type depends on the scrutinee, one branch per constructor of `e`'s type, and each branch produces a result of the motive instantiated at that constructor. The branches must be *exhaustive*. When the motive does not mention the scrutinee, the rule degenerates to the standard non-dependent match where every branch has the same result type `P`; when the motive does mention `x`, each branch's expected type is refined by the branch's constructor. This lets the cross-jurisdictional rule in Section 7.2 return a result type `Pi(_ : EntityContext j). ComplianceVerdict` that varies with the jurisdiction `j`. When the scrutinee itself lives in `Prop`, the side-condition from the `Prop sort` section applies: elimination into `Type_k` is admitted only for singleton propositions, so disjunction and squash proofs cannot be case-analyzed to produce runtime data. In the admissible fragment (Section 5), match is restricted to non-dependent motives over prelude constructor types with known finite variants. The match's effect row is the union of the scrutinee's effects and every branch's effects.

**Structural refinement and a deeper companion finding.** The Rocq mechanization of the typing rule (`T_Match` in `lex/formal/coq/Lex/Typing.v`) types each branch body against `shift 0 arity return_ty` in the branch's extended context. This arity-shifted premise is necessary: without it, weakening at depth `k` produces a goal of the form `shift (k + arity) 1 return_ty` while `T_Match`'s weakened conclusion wants `shift k 1 return_ty`; the two agree only when `return_ty` has no free variables in `[k, k + arity - 1]`, which is not a structural invariant of the unshifted premise. The arity-shifted premise preserves semantics and canonicalises the shift behaviour. The inner-fix theorem has the exact type `weakening_at_match_holds : conv_eq_shift_compat_spec → weakening_at_match_spec`: it removes the former match-specific premise, but it does not

remove the global conversion-shift obligation. A companion structural finding on the operational rule `step_match_ctor_fire` is reported in §5.1: a layered pre-shift of constructor arguments does not achieve shift-equivariance because `subst_args` is a fold whose per-position cutoff is asymmetric to outer-context shifts, and the genuine resolution requires a new parallel multi-substitution primitive in the DeBruijn library. `weakening_property` therefore remains conditional on the single Prop spec `conv_eq_shift_compat_spec`.

**Type annotation:**

```
G |- A : Type_i [empty]    G |- e : A [rho]
------------------------------------------ (Annot)
      G |- (e : A) : A [rho]
```

**Annot** says: an annotation is a machine-checkable assertion that `e` has type `A`; the effect row is preserved unchanged.

**Structural recursion:**

```
G, f:A |- b : A [rho]
---------------------------- (Rec)
G |- fix f : A := b : A [rho]
```

**Rec** says: a recursive definition binds a name `f` that may refer to itself within its own body. This is the full calculus's mechanism for unbounded computation; the admissible fragment (Section 5) excludes it to guarantee termination.

### 4.3 The Effect System

Lex has a row-polymorphic effect system (functions can be generic over which effects they require, enabling code reuse across different effect contexts). An **effect row** is a set of individual effects from the following vocabulary:

| Effect | Meaning |
|---|---|
| `read` | Pure observation of state |
| `write(scope)` | Mutation within a named scope |
| `attest(authority)` | Assertion under an authority |
| `authority(ref)` | Exercise of an authority role |
| `oracle(ref)` | Query to an external oracle |
| `fuel(level, amount)` | Computational fuel consumption |
| `sanctions_query` | Distinguished sanctions check |
| `discretion(authority)` | Exercise of discretionary judgment |

Effect rows form a bounded semilattice under join (+), with the empty row as unit. The ordering is by subsumption: row a is subsumed by row b when every effect in a is also in b. This models effect weakening: a pure function can be used where an effectful one is expected; the reverse use is ill typed.

```
G |- e : A [rho_1]    rho_1 subseteq rho_2
------------------------------------------ (Effect-Weaken)
          G |- e : A [rho_2]
```

**Effect-Weaken** says: if an expression requires effects `rho_1`, it can be used in any context that permits a superset of those effects. A pure function (no effects) can be used anywhere, but a function that performs a sanctions query can only be used where sanctions queries are permitted. Effects only widen, never narrow, you cannot use an effectful function in a context that forbids its effects.

The `sanctions_query` effect is distinguished: the type checker tracks its presence to trigger additional compliance obligations at the call site.

A **branch-sensitive** wrapper `<branch_sensitive> rho` marks effect rows whose join would raise the privilege level. These require an explicit `unlock` eliminator, preventing privilege escalation through branch composition.

The lattice structure provides the algebraic backbone for reasoning about effect composition:

```
effect_join(rho_1, rho_2) = rho_1 union rho_2     (semilattice join)
effect_meet(rho_1, rho_2) = rho_1 intersect rho_2  (meet)
is_pure(rho) iff rho = empty and not branch_sensitive(rho)
```

Join is commutative, associative, and idempotent. The empty row is the unit. These are standard properties of a join-semilattice.

**Theorem (effect monotonicity).** For every effect-derivation tree induced by a typing derivation `G ⊢ e : A [rho]`, and every subderivation corresponding to a subterm `e'` with row `rho'`, the subderivation row is subsumed by the enclosing row: `rho' subseteq rho`. The Rocq target `effect_monotonicity` is Qed-closed in `lex/formal/coq/Lex/PaperMechanization.v`; `Print Assumptions effect_monotonicity` reports closed under the global context.

*Proof.* Project the typing derivation to its finite row-labelled effect tree. In the Rocq development the closed theorem is stated over this extracted tree; the extraction from the implementation typing certificate is a structural projection performed by the elaborator/certificate layer, not a hidden axiom inside the row algebra. A leaf is labelled by its local row. An internal node is labelled by the join of its local row and the rows of its immediate subderivations; n-ary typing rules such as `Match-Dep` and `Defeasible` are encoded by reassociation of binary joins. The proof is structural induction on the subderivation relation. The reflexive case is row-subsumption reflexivity. In the left- and right-child cases, the induction hypothesis gives containment in the immediate child, and the row-join laws include each child row in the parent row;

transitivity gives containment in the enclosing row. The **Fill** rule is the one place an effect row shrinks: fill discharges `{discretion(auth)}` once the hole has been resolved with a PCAuth witness, and the resulting term carries the empty row. This is consistent with monotonicity because the hole's effect is local to the pre-fill derivation; once **Fill** has fired, the unfilled-hole subderivation does not appear in the resulting derivation.

Effect monotonicity is the structural invariant that lets a verifier decide, by inspecting a rule's outer effect row, whether evaluation may encounter discretion, sanctions, or oracle calls anywhere in its execution.

### 4.4 Defeasible Rules

The typing rule for a defeasible rule requires type agreement between the base and all exceptions, Boolean guards, and an effect-row join across base, guards, and exception bodies:

```
G |- A : Type_i    G |- b : A [rho_0]
for each exception (guard_k, body_k, priority_k):
    G |- guard_k : Bool [sigma_k]    G |- body_k : A [rho_k]    priority_k ∈ ℕ
------------------------------------------------------------------------------ (Defeasible)
G |- defeasible b unless (guard_1 => body_1 [p_1] | ... | guard_n => body_n [p_n])
     : A [rho_0 U U_k (sigma_k U rho_k)]
```

**Defeasible** says: a defeasible rule has a base conclusion `b` and zero or more exceptions, each with a guard, an overriding body, and a priority. Three constraints hold simultaneously. Every exception body must inhabit the same type as the base body. Every guard must be of type `Bool`, guards are predicates evaluated to true or false, not arbitrary terms. Priorities are natural numbers drawn from ℕ and participate in the evaluation strategy, not the typing derivation.

The effect row of a defeasible rule is the join (semilattice union) of the base body's effects, every guard's effects, and every exception body's effects. This is the ex-ante effect row. At runtime, the eliminator `defeat(r)` discharges effects from inactive branches. The type system still commits to the union, so an exception-only effect (a discretion hole guarded by a rare condition, or a sanctions query in an exception body) cannot appear pure.

**Proposition (Defeasible effect join).** In any well-typed defeasible term, the resulting effect row is the join of the effect rows of its base body, every exception guard, and every exception body. No branch contributes an effect that is not visible in the outer row.

The elimination form `defeat(r)` reduces a defeasible rule to its resolved value by evaluating guards in descending priority order.

**Simplicial reading (optional).** A defeasible rule with base body `b` and `n` priority-ordered exceptions `(g_1, b_1, p_1), \ldots, (g_n, b_n, p_n)` presents naturally as a semi-simplicial object. Take the vertex set `V = {b, b_1, \ldots, b_n}` and, for each priority-descending chain `p_{i_0} > p_{i_1}`

`> \cdots > p_{i_k} > p_0`, a `k`-simplex recording the partial firing along that chain, with face maps `d_j` given by dropping the `j`-th priority level. When the priorities are strict (no equal-priority ties), the resulting object is an ordered semi-simplicial set in the Eilenberg-Zilber sense (Eilenberg and Zilber 1950; cf. May, *Simplicial Objects in Algebraic Topology*, 1967, §1); when ties are admitted the same construction yields a generalized simplicial set whose equal-priority exceptions collapse to a shared face. The priority evaluator of §7.4 is then the nerve-level map

```
Nrv(r)  --eval-->  Nrv(V_verdict)
```

from the defeasibility nerve `Nrv(r)` into the nerve of the verdict chain used by the Lex presentation, with `eval` factoring through the top-firing body projection. The current admissible fragment requires a total order on exceptions, obtained from `(priority, source-position)`; equal-priority meet is a target extension, not the current mechanized evaluator. The priority evaluator agreement proposition in §7.4 is therefore the strict/total-order pointwise evaluation. This simplicial presentation is optional scaffolding, it does not modify the typing rule or its effect row. Its utility is as a target for later work on persistent audit trails: cross-jurisdictional comparison becomes a simplicial map, and a bridge witness becomes a simplicial homotopy.

### 4.5 Temporal Modals

Lex has two temporal layers: first-class temporal objects and propositions about them. The objects carry effective dates, pack versions, rewrites, repeals, and derived deadlines. The propositions state when facts and obligations hold.

```
G |- r : RuleRef    G |- t0 : Time_0
----------------------------------- (EffectiveDate)
 G |- EffectiveDate(r, t0) : Time_0

G |- r : RuleRef    G |- P : PackType    G |- te : Time_0
G |- EffectiveDate(r, t0) : Time_0    r in P    t0 ⩽ te
------------------------------------------------------- (Rule-Active)
           G |- Active(r, P, te) : Prop

G |- id : PackId    G |- version : PackVersion
G |- rules : Set Rule    G |- effective_date : Time_0
---------------------------------------------------- (Pack)
      G |- Pack(id, version, rules, effective_date) : PackType

G |- P_old : PackType    G |- P_new : PackType
G |- conserves_or_repeals(P_old, P_new) : Prop
G |- preserves_survivor_types(P_old, P_new) : Prop
```

```
G |- P_old.effective_date ⩽ P_new.effective_date : Prop
G |- replay(P_old, P_new) : ReplayWitness
------------------------------------------------------------- (Rewrite)
           G |- Rewrite(P_old -> P_new) : RewriteWitness

G |- rule_ref : RuleRef
---------------------- (Repeal)
G |- Repeal(rule_ref) : RepealWitness

G |- rule_ref : RuleRef    G |- trep : Time_0
------------------------------------------------------------ (Repeal-Effective)
G |- EffectiveDate(Repeal(rule_ref), trep) : Time_0

G |- r : RuleRef    G |- P : PackType    G |- te : Time_0
G |- pi : Active(r, P, te)
------------------------------------------------------------ (Eval-Rule)
        G |- eval(r, P, te) : ComplianceVerdict
```

**EffectiveDate** says: a rule carries a typed application threshold in frozen historical time. **Rule-Active** says: a rule can be evaluated only when the event time is no earlier than its effective date and the rule is present in the selected pack. **Pack** says: a rule pack is itself a typed temporal object, identified by content-addressed rules, version, and effective date. **Rewrite** says: pack evolution is witnessed explicitly rather than inferred from mutable state. **Repeal** says: repeal is a first-class term naming the rule reference it removes. **Repeal-Effective** says: repeal has its own effective date, just like any other legal change. **Eval-Rule** says: the evaluator requires a proof that the rule is active under the chosen pack at the event time. A repeal therefore does not erase prior derivations. It changes which pack can furnish `Active(r, P, te)` for fresh evaluation after the repeal date.

Pointwise temporal modalities:

```
G |- t : Time_i    G |- A : Prop
------------------------------- (At)
     G |- @t A : Prop

G |- t : Time_i    G |- A : Prop
------------------------------- (Eventually)
     G |- diamond t A : Prop

G |- t1 : Time_i    G |- t2 : Time_i    G |- A : Prop
---------------------------------------------------- (Always)
```

```
G |- box[t1, t2] A : Prop
```

**At** says: you can assert that a proposition holds at a specific time, "the company was compliant on 2024-03-15." **Eventually** says: you can assert that a proposition will hold at some point after a given time, "the filing will be submitted after the incorporation date." **Always** says: you can assert that a proposition holds throughout an entire time interval, "the company maintained its registered office service provider from incorporation to dissolution."

`@t A` asserts that `A` holds at time `t`. `diamond t A` asserts that `A` holds at some point after `t`. `box[t1, t2] A` asserts that `A` holds throughout the interval `[t1, t2]`.

The temporal rewrites:

```
G |- t : Time_0
------------------- (Lift)
G |- lift_0(t) : Time_1

G |- t : Time_0    G |- w : RewriteWitness
---------------------------------------- (Derive)
      G |- derive_1(t, w) : Time_1

G |- d : Time_1    G |- t_tol : Time_0
------------------------------------ (Toll)
      G |- Toll(d, t_tol) : Time_1

G |- d : DerivedTime(P)    G |- w : Rewrite(P -> P')
---------------------------------------------------- (Reevaluate)
      G |- reevaluate(d, w) : DerivedTime(P')
```

**Lift** says: any frozen historical time can be treated as a derived legal time, historical facts can feed into legal consequences. **Derive** says: given a historical time and a legal rewrite witness (a record of which statute or amendment produces the derivation), you can produce a new derived legal time. For example, an incorporation date (Time_0) combined with a statutory rule ("file within 28 days") produces a filing deadline (Time_1). **Toll** says: a tolling event consumes an already-derived deadline and produces a new derived deadline. Because the input is `Time_1`, tolling stacks: `Toll(Toll(d, t_1), t_2) : Time_1`. This closes the defect where repeated tolling would otherwise have to re-derive from frozen historical time. **Reevaluate** says: given a derived-time closure and a typed pack rewrite witness carrying replay transport, Lex can compute the post-rewrite consequence without invalidating the original term.

**Lift** says: any frozen historical time can be treated as a derived legal time, historical facts can feed into legal consequences. **Derive** says: given a historical time and a legal rewrite witness (a record of which statute or amendment produces the derivation), you can produce a new derived legal time. Section 3.3 refines this

witness to the pack-aware notation `w_P`. For example, an incorporation date (Time_0) combined with a statutory rule ("file within 28 days") produces a filing deadline (Time_1).

There is no elimination from Time_1 to Time_0. This is an asymmetry by design, not an omission.

**Unindexed temporal proposition sugar.** These forms abbreviate the indexed finite-horizon forms above at the current admissibility horizon; they do not add a second temporal calculus.

```
G |- phi : Prop
------------------- (Eventually-Temporal)
G |- diamond phi : Prop

G |- phi : Prop
--------------- (Always-Temporal)
G |- box phi : Prop
```

The operators `diamond phi` and `box phi` desugar to `diamond t_current phi` and `box[t_start,t_end] phi` for the local finite admissibility horizon. They internalize finite-horizon admissibility claims: an obligation may eventually become due within the rule-evaluation horizon, or hold throughout the admissible local horizon. They are not a closed behavioral temporal logic for workflow traces. A full `until` operator, CTL/TCTL-style branching or clocked trace properties, and their compilation into Op monitors are open obligations. Quantified time remains in the term layer as `Time_0` and `Time_1` objects, effective dates, and tolling events.

Temporal objects combine with tribunal modals to represent authority-relative divergence. If `P_A^1 = Pack(id_A, v_1, rules_A, effective_date = 100)` and `P_B^1 = Pack(id_B, v_1, rules_B, effective_date = 120)`, then for an event `e` at `t = 110` one may derive `G ⊢ v_A : [A] Verdict(e, P_A^1)` and `G ⊢ v_B : [B] Verdict(e, P_B^0)`. No contradiction follows. The verdicts are indexed by different tribunals and different active packs. A bridge witness `w_AB : CanonBridge(A, B, Verdict(e))` is required before any coercion across the two authorities is well-typed.

### 4.6 Tribunal Modals

Following Pfenning and Davies (2001) and the contextual modal refinement of Nanevski, Pfenning, and Pientka (2008), Lex treats the tribunal modal as a type former rather than merely a proposition former. If `A` is a type and `T` is a tribunal, then `[T]A` is the type of `A`-evidence asserted under `T`. As in the temporal fragment, effect rows are omitted here because the modal structure is orthogonal to effect accumulation.

```
G |- A : Type_i    T : Tribunal
------------------------------- (Tribunal-Form)
         G |- [T] A : Type_i
```

The introduction rule is guarded by a typed authority-recognition judgment. Fix an authority theory `Sigma_auth` containing charter clauses, delegation clauses, recognition clauses, and mutual-recognition clauses. Write `T ⊢auth A` when `Sigma_auth` derives that tribunal `T` is recognized to assert terms of type `A`. For this side judgment only, Lex allows an auxiliary preorder `⩽_auth` on authority interfaces. It is not a program-level subtyping relation on Lex terms; it exists only to discharge tribunal side conditions.

The raw bridge syntax is intended to induce a bridge category **Br**(**A**) after quotienting by explicit bridge equality or carrying bicategorical coherence data. Its objects are tribunals and its candidate hom-sets are `CanonBridge(T_i,T_j,A)`. The modal action is intended to be covariant in that bridge orientation: a bridge `b : CanonBridge(T1,T2,A)` gives `coerce[b] : [T1]A → [T2]A`, so `[.]A` is a target indexed action `Br(A) → Type_i`, not a contravariant functor to `Prop`. The category/action laws are exactly the open obligations stated below; before those laws are discharged, this paragraph specifies the intended semantic target rather than a closed theorem. For proposition interfaces `A : Prop`, this specializes to authority-indexed propositions; for general `A : Type_i`, it is a type former over evidence. Tribunal soundness, a verdict cannot be silently reinterpreted between tribunals that disagree, becomes the statement that the relevant hom-set is empty: no bridge witness inhabits `CanonBridge(T_i,T_j,A)`, so the coercion has no term to carry the verdict across. Authorities remain unordered (Section 3.4); the category has no terminal or initial object, and no "universal verdict" functor emerges from the tribunal layer. Any aggregation of verdicts across tribunals happens above this layer, over the tensors that tribunals produce, not over the authorities themselves.

```
charter(T, A) in Sigma_auth
--------------------------- (Auth-Charter)
        T |-auth A

T1 |-auth A    delegate(T1, T2, A) in Sigma_auth
------------------------------------------------ (Auth-Delegate)
                    T2 |-auth A

T2 |-auth A    recognize(T1, T2, A) in Sigma_auth
------------------------------------------------ (Auth-Recognize)
                    T1 |-auth A

T |-auth A0    A0 ⩽_auth A
--------------------------- (Auth-Sub)
        T |-auth A
```

**Auth-Charter** is the base case: a tribunal has authority over the interfaces named in its charter. **Auth-Delegate** propagates authority along a typed delegation edge. **Auth-Recognize** imports an already-derived

authority judgment across a recognition clause; this is where mutual-recognition regimes enter the logic. **Auth-Sub** closes the judgment upward along the auxiliary authority-interface preorder.

```
G |- e : A       T |-auth A
--------------------------- (Tribunal-Assert)
   G |- assert[T](e) : [T]A

G |- m : [T1]A    G |- b : CanonBridge(T1, T2, A)
------------------------------------------------ (Tribunal-Coerce)
              G |- coerce[b](m) : [T2]A

G |- m : [T]A    T |-auth (A -> B)    G, x:A |- e : B@T
------------------------------------------------------ (Tribunal-Open)
             G |- open_T m as x. e : B@T

G |- e : B@T    G |- b : CanonBridge(T, Ambient, B)
--------------------------------------------------- (Tribunal-Adopt)
             G |- adopt[b](e) : B
```

**Tribunal-Assert** is the term-level introduction form: a term becomes tribunal-scoped only when the authority theory derives that the tribunal is recognized for that interface. This blocks laundering an arbitrary term into the modal layer by wrapping it in a tribunal tag. **Tribunal-Coerce** moves a tribunal-scoped term along an explicit bridge witness and nowhere else. **Tribunal-Open** is the local elimination form: one may use the contents of a modal term only to derive a conclusion still marked at tribunal `T`. **Tribunal-Adopt** is the separate bridge-to-ambient rule; an unmodalized ambient conclusion is available only when an explicit adoption bridge is present and is recorded in the receipt/proof trace.

The computation rule is substitution:

```
open_T assert[T](v) as x. e  -→  e[v/x] : B@T
```

for values `v`, provided the typing derivation carries `T ⊢auth (A → B)`. Opening a tribunal assertion is therefore not a bare destructuring step and does not erase authority provenance; ambient use is a later `adopt[b]` step with an explicit bridge.

**Theorem** (**Non-accidental-laundering soundness**)**.** If `G ⊢ assert[T](e) : [T]A` is derivable, then `T ⊢auth A` is derivable.

*Proof.* By induction on the typing derivation `D : G ⊢ assert[T](e) : [T]A`. If the last rule is **Tribunal-Assert**, the second premise is exactly `T ⊢auth A`. No other introduction rule has a conclusion whose head constructor is `assert[T](...)`. Structural wrappers such as explicit annotation preserve

the same immediate subderivation, so the induction hypothesis applies. Hence every derivation of `G ⊢ assert[T](e) : [T]A` contains a derivation of `T ⊢auth A`. ■

Bridge witnesses are generated from bilateral recognition evidence:

```
T |-auth A
------------------------------- (Bridge-Id)
G |- id_T^A : CanonBridge(T, T, A)

T1 |-auth A    T2 |-auth A    mutual_recognize(T1, T2, A) in Sigma_auth
------------------------------------------------------------------------ (Bridge-Mutual)
          G |- bridge[T1 <-> T2, A] : CanonBridge(T1, T2, A)

G |- b12 : CanonBridge(T1, T2, A)    G |- b23 : CanonBridge(T2, T3, A)
------------------------------------------------------------------------------ (Bridge-Comp)
               G |- b23 * b12 : CanonBridge(T1, T3, A)
```

For parallel bridges `b1, b2 : CanonBridge(T1, T2, A)`, define bridge 2-cell equality by extensional equality of their coercion actions:

```
b1 ≡_bridge b2  :⟺  forall m : [T1]A, coerce[b1](m) ≡ coerce[b2](m).
```

**Proposition (Bridge 2-cell congruence).** Bridge equality respects both vertical and horizontal composition.

1. If `alpha : b1 ≡_bridge b2` and `beta : b2 ≡_bridge b3`, then `beta · alpha : b1 ≡_bridge b3`.
2. If `alpha : b12 ≡_bridge b12'` and `beta : b23 ≡_bridge b23'`, then `beta *_h alpha : (b23 * b12) ≡_bridge (b23' * b12')`.

*Proof.* Vertical composition is pointwise transitivity of definitional equality. For horizontal composition, fix `m : [T1]A`. By **Bridge-Comp**, `coerce[b23 * b12](m) ≡ coerce[b23](coerce[b12](m))`. Apply `alpha` to rewrite the inner coercion and `beta` to rewrite the outer coercion, then reassemble the composite with **Bridge-Comp**. ■

**Theorem (canonical bridge bicategory / strict function target).** For each fixed interface `A`, the canonical extensional target takes bridges to ordinary functions between modal carriers and bridge 2-cells to pointwise equality. In this target, bridge equality is an equivalence relation, identity and associativity hold, and whiskering respects 2-cells. `BridgeSemantics.v` proves these facts as `function_bridge_bicategory_laws`.

The objects are tribunals `T`; 1-cells `T1 → T2` are witnesses `b : CanonBridge(T1,T2,A)`; 2-cells `b1 ⇒ b2` are proofs of `b1 ≡_bridge b2`. Bridge equality is the extensional projection used for modal action. Operational certificates still carry the proof-relevant `CanonBridge` witness that justified the coercion. The paper uses the strict notation `b23 * b12` for readability. The strict function target is closed; the richer proof-relevant quotient/strictification theorem for raw bridge syntax and certificate identity remains open.

**Theorem (tribunal modal as canonical bridge action).** For each fixed interface `A`, the canonical assignment is covariant:

```
[-]A : Bridge_2(A) → Type_i
```

with `F_0(T) := [T]A`, `F_1(b) := (m ⟼ coerce[b](m))`, and `F_2(alpha)` the pointwise equality between coercion functions induced by `alpha`. `BridgeSemantics.v` proves modal-action identity, modal-action composition, 2-cell congruence, and the combined `function_bridge_strict_2functor_coherence` theorem for the strict function-bridge target. Presheaf naturality and proof-relevant raw bridge syntax remain adequacy obligations for the full semantic model.

**Theorem (Admissibility lift for authority recognition).** Assume `Sigma_auth` contains finitely many charter, delegation, recognition, and mutual-recognition clauses, and that the auxiliary preorder `⩽_auth` is decidable on the finite set of interfaces appearing in `Sigma_auth` and the query. Then, for every tribunal `T` and every admissible authority interface `A`, derivability of `T ⊢auth A` is decidable.

*Proof.* Let `V` be the finite set of tribunal-interface pairs appearing in `Sigma_auth` together with the query pair and every interface reached by `⩽_auth` during saturation. Define a monotone operator `F : P(V) → P(V)` that adds exactly the pairs justified by one application of **Auth-Charter**, **Auth-Delegate**, **Auth-Recognize**, or **Auth-Sub** whose premises are already present. Because `V` is finite, iterating `F` from the empty set reaches the least fixed point in finitely many steps. Every membership test in one round is decidable because the authority theory is finite and `⩽_auth` is decidable. Soundness follows from the construction of `F`; completeness is by induction on derivations of `T ⊢auth A`. Hence membership of `(T, A)` in the least fixed point decides `T ⊢auth A`. ■

This admissibility lift applies to the side judgment `⊢auth` only. Tribunal modal terms themselves remain outside the admissible term fragment of Section 5 until the stratification conditions for modal reduction are proved.

**Worked ADGM/DIFC mutual-recognition example.** Let `A := FitAndProper(d)` for a director `d` and let the authority theory contain:

`charter(ADGM_FSRA, A)`, `charter(DIFC_DFSA, A)`, `recognize(ADGM_FSRA, DIFC_DFSA, A)`, `recognize(DIFC_DFSA, ADGM_FSRA, A)`, `mutual_recognize(ADGM_FSRA, DIFC_DFSA, A)`.

Then:

```
charter(ADGM_FSRA, A) in Sigma_auth
---------------------------------- (Auth-Charter)
         ADGM_FSRA |-auth A

charter(DIFC_DFSA, A) in Sigma_auth
---------------------------------- (Auth-Charter)
         DIFC_DFSA |-auth A
```

Recognition also gives explicit cross-side derivations:

```
DIFC_DFSA |-auth A    recognize(ADGM_FSRA, DIFC_DFSA, A) in Sigma_auth
------------------------------------------------------------------------ (Auth-Recognize)
                          ADGM_FSRA |-auth A

ADGM_FSRA |-auth A    recognize(DIFC_DFSA, ADGM_FSRA, A) in Sigma_auth
------------------------------------------------------------------------ (Auth-Recognize)
                          DIFC_DFSA |-auth A
```

From bilateral derivability and the mutual-recognition clause we obtain the concrete bridge

```
b_AD : CanonBridge(ADGM_FSRA, DIFC_DFSA, A)
```

by **Bridge-Mutual**, and symmetrically `b_DA`. If `G ⊢ p : A`, then

```
G ⊢ assert[ADGM_FSRA](p) : [ADGM_FSRA]A
```

by **Tribunal-Assert**, hence

```
G ⊢ coerce[b_AD](assert[ADGM_FSRA](p)) : [DIFC_DFSA]A
```

by **Tribunal-Coerce**. If additionally `DIFC_DFSA ⊢auth (A → ComplianceVerdict)`, then

```
G ⊢ open_DIFC_DFSA coerce[b_AD](assert[ADGM_FSRA](p)) as x. verdictOf(x) :
ComplianceVerdict.
```

The example is concrete in the sense relevant to the calculus: the theory names the two regulators, the recognized interface, the bilateral recognition clauses, and the resulting bridge term explicitly.

The surface syntax requires the witness explicitly: `coerce[T1 ⇒ T2](e, w)`. A surface form that elides the witness is not accepted by the elaborator; no default or placeholder witness is supplied, because a coercion without a named bridge witness is the exact failure mode the tribunal modal is designed to prevent.

### 4.7 Typed Discretion Holes

Lex distinguishes three hole constructors:

```
G |- e : A [empty]
------------------------------------------------ (MechanicalHole)
G |- MechanicalHole(h, e) : A [empty]

------------------------------------------------------------------ (DiscretionHole)
G |- DiscretionHole(auth, h, S) : A [discretion(auth)]

------------------------------------------------------------------ (UnsettledHole)
G |- UnsettledHole(dom, h) : A [unsettled(dom)]

G |- e : A    G |- w : PCAuth(auth, h, e, request(h,e))
---------------------------------------- (Fill)
      G |- fill(h, e, w) : A   [empty]
```

`MechanicalHole` marks a site whose answer is already within the core of settled meaning. It carries the empty effect row and reduces immediately to its mechanically derived witness `e`. `DiscretionHole` marks Hart's penumbra: the term is well typed, but it emits `discretion(auth)` and cannot reduce further until an authorized human supplies a fill. `UnsettledHole` emits `unsettled(domain)`. This is a different effect: no existing authority may discharge it with a `PCAuth` witness, because the present rule pack has no disposition to apply.

A `DiscretionHole` of type `A`, authorized by `auth`, with optional scope `S`, has type `A` and produces the `discretion(auth)` effect. An `UnsettledHole` of type `A` in domain `dom` produces the `unsettled(dom)` effect. Both forms are well typed in the full calculus and both are rejected by the admissible fragment (Section 5), but for different reasons: one requires an authorized judgment; the other requires lawmaking.

Discretionary filling:

```
G |- e : A    G |- w : PCAuth(auth, h, e, request(h,e))
---------------------------------------- (Fill)
      G |- fill(h, e, w) : A [empty]

G |- e : A    G |- w : PCAuth(auth, h, e, request(h,e))
---------------------------------------- (RecordFill)
G |- RecordedFill(h, e, w) : FilledDiscretionHole(h)

G |- fills : List(FilledDiscretionHole(h))
---------------------------------------- (Precedent)
G |- Precedent(h, fills) : PrecedentChain(h)
```

```
G |- f : FilledDiscretionHole(h)
G |- e' : A    G |- w' : PCAuth(higher_auth, h, e', request(h,e'))
------------------------------------------------------ (Appeal)
G |- Appeal(f, higher_auth, e', w') : AppealedFill(h, f, higher_auth)

G |- r : PackRewrite(dom, P, P')    G[P'] |- e : A [empty]
------------------------------------------------------------ (PackRewrite)
G[P] |- settle(h, r, e) : A [empty]
```

**Fill** says: filling a discretionary hole requires a value of the correct type together with a proof-carrying authorization witness (PCAuth) attesting that the filler has the authority to supply it. Once filled, the `discretion` effect is discharged and the result is pure, so downstream computation can proceed without further human input.

The filling `e` must have type `A` (the hole's type). The witness `w` must prove that the filler has authority `auth` for hole `h`, value `e`, and the current fill request. The filled result is pure; the `discretion` effect has been discharged by the witness. **RecordedFill** materializes the audit object that later precedent chains and appeals consume. **Precedent** packages a typed chain of past fills for consultative use; later fillers may cite it but are not bound by it. **Appeal** records a higher-authority re-fill, and downstream verdicts depending on `f` are tagged `Appealed` until re-derived. **PackRewrite** is the only eliminator for `UnsettledHole`: it requires a typed witness that the governing rule pack has evolved from `P` to `P'` and re-runs the derivation under `P'`. No individual `PCAuth` witness can discharge `unsettled(dom)`.

The filling `e` must have type `A` (the hole's type). The witness `w` must prove that the filler has authority `auth` for the exact judgment `(h, e)` and the exact request, pack, and context digests; changing the value or the context changes the witness type. The filled result is pure, the `discretion` effect is discharged.

For holes whose policy requires a quorum, the witness type is refined to `PCAuth_k(auth, h, e, request(h,e))`, meaning that the same value-indexed witness additionally certifies a `k-of-n` threshold over the exact value `e` and request context.

Revocation is typed rather than ambient. We use `Depends(j, c)` for the transitive dependency judgment on derivation traces: it holds when credential `c` occurs anywhere in the justification tree of judgment `j`. We also write `Tagged(A, s)` for a value of type `A` tagged with status `s in {Clean, Tainted, Invalid}`.

```
G |- c : Credential
------------------------ (Revoke)
G |- Revoke(c) : RevokedCred

G |- j : A    G |- r : RevokedCred
Depends(j, subject(r))    fill_time(j) < revoke_time(r)
```

```
------------------------------------------------------ (Taint)
G |- taint(j, r) : Tagged(A, Tainted)

G |- j : A    G |- r : RevokedCred
Depends(j, subject(r))    revoke_time(r) ⩽ fill_time(j)
------------------------------------------------------ (Invalidate)
G |- invalidate(j, r) : Tagged(A, Invalid)
```

**Revoke** produces a typed revocation artifact. **Taint** marks a historical judgment whose supporting credential was revoked only after the fill/admission time; the judgment remains in the record but must be re-evaluated or flagged downstream. **Invalidate** marks the stronger case in which revocation predates or coincides with fill/admission time, so the witness should never have been accepted.

Delegation is explicit and depth-bounded. A root authority credential `AuthorityCred(auth, p, S)` says that principal `p` holds authority `auth` over scope `S`. `DelegateCred(auth, p, q, S, d)` says that principal `p` delegates that authority to principal `q` over sub-scope `S` with residual depth `d`.

```
G |- c : AuthorityCred(auth, p, S)    G |- q : Did    G |- S' subseteq S    d ⩾ 0
------------------------------------------------------------------------------- (Delegat
G |- delegate(c, q, S', d) : DelegateCred(auth, p, q, S', d)

G |- delta : DelegateCred(auth, p, q, S, d+1)    G |- r : Did    G |- S' subseteq S
------------------------------------------------------------------------------- (Delega
G |- delegate(delta, r, S', d) : DelegateCred(auth, q, r, S', d)
```

**Lemma (delegation does not expand authority).** If `delta : DelegateCred(auth, p, q, S', d)` is derivable from a root credential over scope `S`, then `S' subseteq S` and every delegation chain ending in `delta` has length at most the root depth bound. Hence a delegated signer can exercise only authority already present at the root and cannot increase delegation depth.

*Proof sketch.* By induction on the delegation derivation. **Delegate-Root** imposes the scope inclusion `S' subseteq S` directly. **Delegate-Step** preserves inclusion by premise and strictly decreases residual depth from `d+1` to `d`. Transitivity of subset across the chain yields the scope claim, and the strictly decreasing depth yields the bound.

```
G |- e : A    G |- W : PCAuth_k(auth, h, e, request(h,e))
G |- fresh(id(h))    G |- required(h) = k
G |- W.signers subseteq committee(h)    G |- |committee(h)| ⩽ n_max
------------------------------------------------------------------ (Fill-Quorum)
                G |- fill_k(h, e, W) : A   [empty]
```

**Fill-Quorum** is the multi-signer form. At hole introduction, a committee-policy `DiscretionHole` carries two pieces of pack-data: a declared committee `committee(h) : Finset(Did)` (the finite set of principals whose joint determination is legally required) and a threshold `required(h) : Nat`. The default is a singleton committee with `required(h) = 1`; committee holes specify both at introduction. A witness `W : PCAuth_k(auth, h, e, request(h,e))` is well-formed only when `W.signers subseteq committee(h)`, `|W.signers| ⩾ k`, and `k = required(h)`. The value index `e` is shared: every signer commits to the same verdict and the same fill-request, pack, and proof-context digests. Duplicate signer counting is ruled out by `DistinctSigners(signers)` combined with the subset constraint; replay into a different hole or request is ruled out by the signed canonical payload. The deployment bound `n_max` of the forgery reduction is the type-level upper bound on `|committee(h)|` under the deployed authority pack, not a reduction-side slack parameter; the reduction simply exposes it as the slot-guessing denominator.

### 4.8 Delegation Chains

A single authority rarely signs every discretionary decision itself. A regulator delegates signing authority to specific officers; those officers may further delegate to named deputies within documented scope limits. Lex makes this chain typed and bounded.

```
, Delegation types
Delegate(a : Authority, s : Authority, scope : Scope) : DelegationEdge
DelegationChain(root : Authority, leaf : Authority, d : Nat) : Type

                            (Delegate-Base)
G |- refl(auth) : DelegationChain(auth, auth, 0)


G |- c : DelegationChain(auth, s, d)
G |- e : Delegate(s, s', sigma)    d + 1 ⩽ d_max
------------------------------------------------- (Delegate-Step)
G |- ext(c, e) : DelegationChain(auth, s', d+1)
```

**Delegate-Base** and **Delegate-Step** introduce and extend a delegation chain. A root authority has a reflexive chain to itself of depth 0. A chain of depth `d` from root `auth` to intermediary `s` extends to a chain of depth `d+1` to `s'` given a signed delegation edge `s → s'` scoped by sigma, provided the new depth does not exceed a pack-declared bound `d_max` (default 4). The depth bound is a static side-condition: a chain exceeding `d_max` is not well-typed, so the rule logic cannot witness a credential beyond its declared administrative depth.

A delegation edge `Delegate(a, s, scope)` is itself an attested record:

```
Delegate(a, s, scope) = {
  issuer    : Did,             , a's signing principal
```

```
  delegate  : Did,             , s's principal
  scope     : Scope,           , restriction on what s may sign under a
  issued_at : Time_0,
  stamp     : BulletinStamp,
  signature : Ed25519Sig(issuer,
                bind_del(a, s, scope, issued_at))
}
```

The chain's `chain_digest` (referenced by the binding hash in `PCAuth`) is the Merkle root of the ordered list of edge signatures. The bound `d_max` and the scope intersection `sigma_0 cap sigma_1 cap ... cap sigma_d` (intersected along the chain in the meet-semilattice of §3.5, so the result is again a decidable downward-closed predicate) jointly determine what a leaf signer may authorize: the `scope_ok` field of a PCAuth witness must have its predicate entailed by the chain-intersected scope.

Category structure, stated once. The collection of depth-bounded delegation chains forms the hom-sets of a 1-category `Deleg` whose objects are principals and whose morphisms `p -→ q` of depth `d` are the `DelegationChain(p, q, d)` terms: `refl` is the identity at each object, and `ext` is composition with depth strictly incremented. Chains of depth greater than `d_max` are not morphisms of `Deleg` under the deployed pack. Scope tightening `(S, p -→ q) -→ (S' entailed by S, p -→ q)` is a faithful functor from `Deleg` into the slice `Deleg / Scope` of scoped chains; the scope semilattice of §3.5 is the object half of this structure. Revocation and pack-rewrite are not inverses in `Deleg`: they act on the bulletin lattice (§4.9, §4.10) and mark chain morphisms as `Tainted` or `Invalid` without reversing the arrow. A 2-category refinement in which scope-tightening is itself a 2-cell, discussed in §13, would align `Deleg` with the recognition 2-category used for tribunal bridges (§4.6); the present paper commits only to the 1-category structure and uses it to state the soundness proposition below.

**Proposition (Delegation soundness).** If G |- w : PCAuth(auth, h, v, r) is well-typed with a chain of depth d <= d_max, then there exists a sequence of principals p_0, p_1, ..., p_d with p_0 = auth and p_d = w.signer, each link signed by its predecessor, and the chain's scope intersection contains h.scope. The proof is by structural induction on the chain; the base case is **Delegate-Base** (the root signs its own witnesses directly), and the step case preserves the invariant by construction of **Delegate-Step**.

### 4.9 Revocation and Temporal Tagging

Credentials outlive any single judgment. An officer who holds an authorized-signer credential at `t_1` may have that credential revoked at `t_2 > t_1` for cause. Verdicts derived from a filling at `t_1` remain in the audit record; the question is how to characterize their status once the revocation is recorded. Lex fixes this characterization by types.

```
, Revocation artifact
Revoke(c : Credential) : RevokedCred =
```

```
  { target    : Credential,
    reason    : RevocationReason,
    revoked_at: Time_0,
    stamp     : BulletinStamp(t_r),
    signature : Ed25519Sig(issuer_of(c),
                  bind_rev(c.id, t_r, reason))
  }

, Status tag on a derived verdict
VerdictStatus ::= Honored | Tainted | Invalid

G |- w : PCAuth(auth, h, v, r)
NOT exists r. G |- r : Revoke(w.role) /\ r.stamp.t_r ⩽ evaluator_bulletin_position
------------------------------------------------------------------------------ (Status-Honored)
        G |- tag(w) = Honored

G |- w : PCAuth(auth, h, v, r)
G |- r : Revoke(w.role)    w.stamp.t < r.stamp.t_r
------------------------------------------------- (Status-Tainted)
                G |- tag(w) = Tainted

G |- w : PCAuth(auth, h, v, r)
G |- r : Revoke(w.role)    r.stamp.t_r ⩽ w.stamp.t
------------------------------------------------- (Status-Invalid)
                G |- tag(w) = Invalid
```

The three status tags partition verdicts derived from a witness according to the earliest authoritative revocation for the credential visible in the evaluator's bulletin view and the bulletin-ordered comparison of the witness's timestamp `t_w` and the revocation's timestamp `t_r`. **Status-Honored** is the default: no revocation for the credential is visible at or before the evaluator's current bulletin position. **Status-Tainted** marks a witness whose credential was revoked after the filling: the witness was valid at the moment of filling, and the verdict is a historical fact, but its continued authority downstream is a policy question. **Status-Invalid** marks a witness whose credential was already revoked before the bulletin time of the witness itself: the witness should never have been accepted, and verdicts derived from it are void *ab initio*.

```
, Downstream evaluation consumes tagged verdicts
match tag(w)
  | Honored => accept(verdict)
  | Tainted => flag(verdict, revoked_after: r.revoked_at)
  | Invalid => reject(verdict, reason: "revoked before filling")
```

The bulletin timestamp is non-negotiable as an append-only log position: both `w.stamp` and `r.stamp` are anchored into the same public bulletin, so their log ordering is a total order a third party can reconstruct. The bulletin does not prove real-world event time by itself; it proves that, once an entry is anchored under the publication policy, the operator cannot later change its Merkle position without detection. The temporal comparison is visible in the derivation trace and reproducible from the bulletin plus the deployment's submission policy.

**Proposition (Revocation determinism).** For every well-typed witness `w` and evaluator bulletin view, select the earliest authoritative revocation for `w.role` visible in that view, if one exists. Exactly one of **Status-Honored**, **Status-Tainted**, **Status-Invalid** applies. The proof is by absence of a visible revocation or by trichotomy on the selected revocation's bulletin-ordered timestamp against the witness timestamp.

Status lattice, stated once. The full verdict-status lattice is the product poset `{Clean, Tainted, Invalid} x {Live, Appealed}` with coordinatewise order `Clean < Tainted < Invalid` and `Live < Appealed`. The trichotomy above is the projection onto the revocation axis; it is deterministic because the bulletin gives a total order on stamps. Appeal (§3.5, §4.7) is orthogonal: a verdict can be `(Tainted, Appealed)` (the witness's credential was later revoked and a higher authority issued a re-fill) or `(Clean, Appealed)` (the witness remains honored and a higher authority nonetheless re-filled) without either coordinate collapsing the other. The section's `Status-Honored/Tainted/Invalid` rules and §3.5's `Appeal` rule therefore fire on disjoint axes and compose by product. The richer lifecycle coordinates discussed in §13.9 (speech-act force, acknowledgment, affirmance) would extend this lattice with a third factor; the present paper fixes only the two orthogonal factors already supported by the calculus.

### 4.10 Bulletin-Anchored Timestamps

The public bulletin that anchors `BulletinStamp` is the Haber-Stornetta (1991) append-only Merkle log: every anchored entry is a leaf of a growing Merkle tree whose roots are periodically published (canonical implementation: certificate transparency logs or an equivalent public bulletin service). A `BulletinStamp(t)` consists of: (a) the leaf index `t` (an integer position in the append-only log, serving as the public time coordinate), (b) a Merkle inclusion proof from the leaf to a published root, and (c) a signed root attestation from the bulletin operator.

```
BulletinStamp(t) = {
  position  : Nat,             , Merkle-log leaf index
  inclusion : MerklePath,      , path from leaf to root
  root_sig  : Ed25519Sig(bulletin_operator, root_at_epoch(t))
}
```

A witness's `stamp` binds the witness into a total order over the log's leaves. Two witnesses on the same bulletin have comparable timestamps; witnesses on distinct bulletins require a declared `BulletinInterop`

witness specifying how to compare. The Haber-Stornetta construction predates blockchain timestamping by eighteen years and is the canonical reference for digital timestamping; its security reduces to collision-resistance of the underlying hash and to append-only integrity of the operator.

### 4.11 Subtyping and Type Inference

Lex has no subtyping relation. Two types are either definitionally equal (up to beta and zeta reduction in the admissible fragment) or they are distinct. A `ComplianceVerdict` is not a subtype of `Nat` despite both being finite enumerations; a `SanctionsResult` is not a subtype of `ComplianceVerdict` despite the semantic proximity of `Clear` to `Compliant`. Where conventional languages use subtyping to allow a caller to supply more specific arguments or a callee to return more general results, Lex uses explicit coercion: if a `SanctionsResult` needs to inform a `ComplianceVerdict` computation, the rule author writes the coercion and it appears in the derivation trace. The design cost is verbosity; the gain is auditability. Every type-level operation is visible in the derivation; no structural conversion happens implicitly.

Type inference is bidirectional and limited. Lambda abstractions require annotated domains at introduction; `let` bindings may be either annotated or inferred from their initializers; pattern-match return types are required in any branching position where the branches could produce structurally distinct types. The bidirectional discipline is standard for dependent type theories and keeps checking decidable within the admissible fragment. A full unification-based inference algorithm for Lex, which would reduce annotation burden at the cost of standard complications around higher-order unification and effect-row polymorphism, is open work noted in Section 13. The choice between annotation and inference is a productivity concern, not a soundness concern; the type system's guarantees hold under either discipline.

### 4.12 Sanctions Dominance

```
G |- p : SanctionsNonCompliant(entity)
------------------------------------------ (Sanctions-Dominance)
G |- sanctions-dominance(p) : bottom_comp

G |- e : bottom_comp    G |- A : Type_i [empty]
---------------------------------------------- (Bottom-Elim)
       G |- absurd(e, A) : A [empty]
```

`bottom_comp` is the empty type at the value level (placed in `Type_0`, not `Prop`), distinct from the proof-irrelevant bottom of the propositional fragment. The placement is load-bearing: a proof-irrelevant bottom is erased at runtime and cannot derive the hard block from the typing; a value-level empty type makes the block a property of the calculus.

**Sanctions-Dominance** says: from a proof of sanctions non-compliance, the rule produces an inhabitant of `bottom_comp`. No ordinary introduction form creates a closed well-typed term of type `bottom_comp`; the only intended source is the Sanctions-Dominance constructor itself. **Bottom-Elim** (*ex falso quodlibet*)

lets any type be inhabited once a witness of `bottom_comp` is in scope, which is precisely the behavior a hard block requires: every downstream derivation typechecks against any expected type but is never reachable. The target terminality theorem is that, before Bottom-Elim is applied, a `bottom_comp` inhabitant can enter a closed derivation only through the sanctions constructor; the paper treats that as a proof obligation unless the cited mechanization closes the corresponding constructor-inversion lemma.

A proof of sanctions non-compliance therefore absorbs the surrounding computation: no defeasibility exception, tribunal coercion, or discretion hole can override it. The runtime sanctions preflight is an *optimization* of this typing behavior, a short-circuit that halts evaluation before `bottom_comp` is constructed, not an independent enforcement mechanism.

---

## 5. The Admissible Fragment

The full core calculus is undecidable. General recursion (`fix`), unrestricted match on user-defined inductive types, and unfilled discretion holes all prevent the type checker from terminating. This is expected, the full calculus is expressive enough to encode arbitrary computation.

For practical use, Lex defines an **admissible fragment**: a syntactic subset whose admissibility predicate is bounded and whose type-checking target becomes decidable only after the finite checks below and the metatheory gates named in §13 are closed. The first check is now Qed-closed in the public Rocq development; the remaining two are explicit residual obligations at the rule-schema or oracle boundary:

1. *Decidable equality on syntax.* The mutually recursive `Term`/`Branch`/`Exception` carriers admit structural decidable equality. `lex/formal/coq/Lex/Syntax.v` now defines `term_eq_dec`, `branch_eq_dec`, and `exception_eq_dec` by a mutual structural decision procedure; `Print Assumptions` reports each closed under the global context. This removes the former equality axioms from the admissibility trusted base.
2. *Decidable exhaustiveness of match.* The typing rule `T_Match` does not structurally enforce that branches cover every constructor of the scrutinee's inductive type; §5.1's progress theorem is Qed-closed *conditional on* a Prop spec `match_exhaustiveness_property`. The stronger formal target is now named `match_coverage_property`: every value reduct of the scrutinee must be covered by the branch list. Rocq closes `match_coverage_implies_exhaustiveness` and `progress_from_match_coverage : confluence_property → match_coverage_property → progress_property`; admissibility still must construct that coverage certificate from finite constructor coverage and confluence remains a separate premise until the diamond closes.
3. *Decidable verification of filled holes.* `VerifyPCAuth(W, h, v, r, t_check)` is total once the bulletin log of §4.10 is queriable at the fill/admission check time `t_check`: canonical payload signature check (decidable), quorum check (decidable on the finite `committee(h)` of §4.7), linked-timestamp

check (decidable modulo the bulletin oracle), credential non-revocation at `t_check` (decidable on a finite bulletin snapshot), delegation-depth bound (decidable against the pack-declared `d_max` of §4.8). The external bulletin query is an oracle obligation, not a defect of the type system; it is named here rather than left implicit in §4.10.

Under these checks, the admissible fragment accepts:

- **Var**, variable references
- **Sort**, with resolved levels (no level variables)
- **Lambda**, lambda abstractions
- **Pi**, dependent function types (pure only, no effect rows)
- **App**, application
- **Annot**, type annotations
- **Let**, let bindings
- **Bounded quantifiers** (`forall x : A in C. P(x)`, `exists x : A in C. P(x)`, `exists! x : A in C. P(x)`, and `choose`), when `C` is a finite prelude collection and the predicate is admissible
- **Constant**, references to names in the global prelude signature
- **Defeasible**, when all sub-terms are admissible
- **Match**, restricted to prelude constructor types (`ComplianceVerdict`, `Bool`, `Nat`, `SanctionsResult`, `ComplianceTag`) with known finite variants

The admissible fragment rejects the following forms, each for a specific reason:

- **Rec** (general recursion): permits non-terminating reduction, directly defeating decidability.
- **Unfilled Hole**: an unfilled discretion hole produces the `discretion(auth)` effect, which cannot be discharged without a PCAuth witness. Admitting unfilled holes would require the type checker to reason about effect discharge obligations that may never be met.
- **General Sigma/Pair/Proj** (dependent pairs): general Sigma types introduce existential witnesses. Without inductive metadata (which the current prelude does not provide for pairs), the type checker cannot determine the type of a projection without evaluating the witness, which may require arbitrary computation. Sigma types are a natural extension target once the prelude provides inductive eliminators for pair types. The bounded existential of Section 4.2 is admitted separately because it compiles to finite search over an explicit collection rather than to unrestricted witness-carrying pairs.
- **All modal forms** (ModalAt, ModalEventually, ModalAlways, ModalIntro, ModalElim): modal operators introduce possible-world semantics where type-checking requires reasoning about propositions indexed by time or authority. The decidability argument requires that reduction terminates uniformly, not conditionally on which world is inhabited. Extending admissibility to include modals requires a stratification argument ensuring that modal operators do not introduce dependency cycles (Section 13).

- **Pi_meta**: higher-order quantification over rules ranges over the open-ended rule space rather than a finite data collection. Admitting it would require reflection over rule syntax and metadata, which falls outside the bounded-reduction argument.
- **SanctionsDominance**: produces the value-level empty type `bottom_comp`, whose elimination admits a term of any type. This breaks the correspondence between admissibility and fuel-bounded WHNF reduction, because any context around a `sanctions-dominance` subterm can be typed at any type without its reduction terminating. The runtime sanctions preflight short-circuits evaluation before the `bottom_comp` term is constructed.
- **PrincipleBalance**, **Unlock**: require reasoning about effect privilege levels that the admissible fragment does not track.
- **Temporal coercions** (Lift0, Derive1): rejected for the same stratification reason as modals. `Derive1` in particular embeds a pack-relative witness `w_P` into its `Time_1` output, so admitting it inside the decidable fragment would require the admissibility predicate to track pack-indexed dependency; §3.3 states the cross-pack commutation obligation this entails (the modal-and-temporal-subsystem conjecture) as an open problem. `Lift0` is rejected for the narrower reason that its admissibility closure would force `Derive1` to be admissible as well, since every admissible consumer of a `Time_1` must be closed under substitution of a `lift_0`-derived argument. The companion Op paper's admissibility gate (§8.1) excludes `Lift0` and `Derive1` from the Op compilation target for the same reason, not an independent one: until the §3.3 conjecture is closed, neither layer has a mechanism for transporting a pack-relative `Time_1` witness through its reduction semantics.
- **Content-addressed references** and **Literals**: content-addressed references require hash resolution, which is an external effect. Literals are syntactic sugar that the elaborator resolves before admissibility checking; they do not appear in the core AST.

The logic of the boundary is this: the admissible fragment includes every form whose reduction behavior is bounded either by the term's syntactic size or by the cardinality of an explicit finite collection, and excludes every form that requires either (a) unbounded computation (Rec), (b) external resolution (Hole, content-addressed references), (c) possible-world or reflective reasoning (modals, temporal coercions, Pi_meta), or (d) bottom-type elimination (SanctionsDominance). The goal is a fragment where weak head normal form (WHNF) reduction is a total function from terms to values, with no appeals to oracles, authorities, or unbounded search.

**Reduction discipline.** The Lex small-step reduction relation `→_L` is the compatible closure of the following head-redex rules, collected here so §10's adequacy statement has a named target:

- *beta*: `(lam x : A. b) a  →_L  b[a/x]`
- *zeta*: `let x : A := e in b  →_L  b[e/x]`
- *iota* (finite match): `match C_i(xs) return P with ... | C_i(xs) ⇒ b_i | ...  →_L  b_i[xs]` when the scrutinee is in constructor form

- *defeat*: `defeat(defeasible b unless (g_1 ⇒ b_1 [p_1] | ... | g_n ⇒ b_n [p_n]))(c) →_L v(c)`, with `v(c)` the §7.4 priority-evaluator output: sort firing exceptions by `(priority descending, source-position ascending)`, select the first firing body, and return the base body if none fires. Equal-priority meet is a target extension for jurisdictions whose rule packs explicitly choose a meet semantics; it is not the default evaluator of this paper.
- *tribunal-open*: `open_T assert[T](v) as x. e →_L e[v/x]` for values `v`, provided `T ⊢auth (A → B)` is derivable in `Sigma_auth`
- *tribunal-coerce-id*: `coerce[id_T^A](m) →_L m`
- *tribunal-coerce-comp*: `coerce[b_{23} * b_{12}](m) →_L coerce[b_{23}](coerce[b_{12}](m))`
- *fill*: `fill(h, e, w) →_L e` when `w : PCAuth(auth, h, e, request(h,e))` verifies at the fill/admission check time `t_check`
- *reevaluate*: `reevaluate(derive_1(t_0, w_P), omega) →_L derive_1(t_0, w_{P'})` for the metalevel pack-replay of §3.3
- *sanctions-absurd*: a term containing `absurd(sanctions-dominance(p), A)` does not reduce; the enclosing computation is terminally blocked by the runtime sanctions preflight before `bottom_comp` is constructed

The compatible closure lifts `→_L` through every constructor's congruence rules (the forty-five binder-congruence rules in `Lex/Typing.v`, with the parallel-reduction companion `par` in `Lex/Confluence.v`). Within the admissible fragment: there is no recursion (`Rec` is excluded); every match is on a finite prelude type with known constructors, so branching is bounded; `Let` unfolds in a single zeta-step, and the number of let-bindings is bounded by the term size; application produces a beta-step, which the evaluator performs under a finite fuel counter and a finite substitution size limit; bounded quantifiers expand to conjunction, disjunction, uniqueness checks, or witness search over an explicit finite enumeration. Conversion is defined as syntactic equality up to weak head normal form under `→_L`: two types are definitionally equal when their WHNFs agree structurally. This is weaker than full beta-eta conversion. The implementation target is fuel-bounded WHNF conversion for the admissible fragment; the closed result below is the administrative WHNF kernel. Extending that bound through beta, finite match, bounded quantifiers, and any future modal, temporal, or discretion admissions remains part of the full metatheory obligation. The adequacy theorem of §10 targets exactly this relation `→_L` once that obligation is discharged.

**Theorem (bounded administrative WHNF).** For the typed administrative weak-head fragment of values, lambdas, annotations, and head lets, `PaperMechanization.v` defines an explicit head-step measure `whnf_head_steps` and proves `administrative_whnf_bounded_reduction`: every closed admissible well-typed administrative term reaches a weak-head-normal value using fuel bounded by `term_size(t) + let_depth(t)`. The theorem is Qed-closed and `Print Assumptions administrative_whnf_bounded_reduction` reports closed under the global context.

The full-calculus version remains part of the broader metatheory: extend the same fuel argument through beta with bounded substitution size, finite match, bounded quantifiers over explicit collections, and the currently excluded modal/temporal/discretion forms only after their stratification and adequacy obligations are closed.

**Conjecture (SN-admissible).** For every closed admissible well-typed term `t : A`, every reduction sequence from `t` terminates (strong normalization).

This is stronger than bounded WHNF reduction: bounded WHNF reduction is a property of the fuel-bounded evaluation strategy, while strong normalization is a property of the calculus itself. The conjecture is open. The flat admissible fragment without Pi-types, modals, recursion, or discretion holes is mechanized as strongly normalizing in Coq (see the formalization status below); extending this to the full admissible fragment remains the primary metatheoretic gap. Decidability of type-checking in the full admissible fragment still depends on the full-calculus WHNF and conversion metatheory together with the bidirectional checking discipline: the checker uses fuel-bounded WHNF equality and rejects any term that exhausts its fuel budget.

**Mechanization status (overview).** Paper-level statements and support kernels have named Rocq targets across the `lex/formal/coq/Lex/` mechanization, with `PaperMechanization.v` carrying the direct correspondence for original abstract targets and dedicated files carrying the finite-algebra and metatheory closures. Four statements in `PaperMechanization.v` now close with `Qed`: `defeasible_effect_join`, `effect_monotonicity`, `finite_observation_event_union_bound`, and `administrative_whnf_bounded_reduction`. The last two are support kernels for stronger paper obligations, not claims that the full cryptographic reduction or full admissible-calculus WHNF theorem is closed.

The bridge equations `bridge_composition` and `bridge_identity_units` remain Qed-closed reflexivity witnesses for simple composition equations. The proof-relevant bridge target is now typed in `lex/formal/coq/Lex/BridgeSemantics.v`; its canonical strict function model closes `BridgeEq` equivalence, identity, associativity, whiskering, modal action identity/composition, 2-cell congruence, and the combined `function_bridge_strict_2functor_coherence` theorem. The same file now names proof-relevant provenance and partial bridge admission layers so the strict extensional target is not confused with certificate extraction or failed admission. The richer raw-syntax quotient/strictification and presheaf naturality obligations remain open. The other Qed-closed entries (temporal non-regression, re-evaluation soundness, verdict Heyting algebra, receipt locality, structural PCAuth quorum extraction, admission-envelope locality, bounded-quantifier admissible expansion, priority-evaluator normalized-meet agreement, tensor/propagation alignment, substitution metatheory) live in their own dedicated files (`TemporalStratification.v`, `PackReevaluation.v`, `VerdictHeyting.v`, `ReceiptAlgebra.v`, `PCAuthQuorum.v`, `AdmissionEnvelope.v`, `BoundedQuantifiers.v`, `BoundedQuantifierAdmissibility.v`, `DefeasibilityPriority.v`, `TensorAlignment.v`, `PropagationAlignment.v`, `DeBruijn.v`, `Typing.v`). The complete per-statement enumeration of

every theorem, proposition, lemma, and conjecture in this paper, with each one classified as Qed-closed, conditional, open, or self-declared conjecture, appears as the **Mechanization Status** appendix at the end of this section. The current ledger count is sixteen Qed-closed entries out of twenty-seven non-conjectural entries, with support kernels explicitly marked where the stronger full obligation remains open.

- **Core scaffold.** An inductive formalization of the admissibility predicate, level resolution, hole authorization, tribunal coercion, temporal lift, summary compilation, and the derivation certificate. Both directions of decidability for admissibility are proved constructively. Level non-self-application is proved (no `Lt L L` inhabitant). Hole authorization is proved in the forward direction (existence of a `PCAuth` witness implies the signer matched the authority). Temporal lift is total; temporal retract is not expressible. One theorem, `principle_balancing_terminates`, establishing the decidability of priority-graph acyclicity, is closed classically via `excluded_middle_informative` from Coq's `ClassicalDescription`; a constructive Tarjan/Kosaraju decision procedure would replace the classical step without changing the theorem statement.
- **Flat admissible strong normalization.** A separate file proves SN for the flat admissible fragment (variables, constants, accessors, let, match, defeasible with non-empty exception lists, and affine lambda) under call-by-value reduction. The proof is a well-founded induction on a size measure using an affine substitution lemma. `Print Assumptions flat_admissible_sn_ext` reports "Closed under the global context", the proof is fully constructive and uses no axioms beyond Coq's standard inductive constructions.
- **Rocq metatheory repair tree.** The companion development in `lex/formal/coq/Lex` tracks the de Bruijn substitution calculus, typing, admissibility, and the progress skeleton used by the Rust implementation. The explicit `Admitted` count across all `.v` files tracked in `_CoqProject` is zero, and the default build surface has no live `admit` tactic and no top-level `Axiom`. Weakening is Qed-closed conditional on `conv_eq_shift_compat_spec`; progress is Qed-closed conditional on `confluence_property` and `match_exhaustiveness_property`, with the stronger coverage route `confluence_property` → `match_coverage_property` → `progress_property` now Qed-closed as `progress_from_match_coverage`; substitution metatheory for binding forms is discharged by `shift_subst_commute_ws` / `subst_subst_ws` / `shift_subst_commute_above`. Confluence, preservation for the full admissible fragment, and admissible-fragment progress in its unconditional form remain open; the gating obligation is the `step_match_ctor_fire` parallel-substitution primitive described in §4.2 and at the end of this section.

What remains open in the mechanization: preservation for the full admissible fragment with Pi-types, effect rows, and discretion holes; unconditional progress beyond the current `confluence_property` premise and the still-unconstructed `match_coverage_property` certificate; confluence through the open `par_diamond_spec`; shift-compatibility for `step_match_ctor_fire` through the open `conv_eq_shift_compat_spec`; and strong normalization beyond the flat affine fragment. The standard

binding substitution lemmas are not open at this level: the relevant shift/substitution commutations are Qed-closed in the current De Bruijn development. These remaining obligations are acknowledged as metatheoretic frontier statements rather than claimed. This paper's contribution is the design of the logic and the identification of the admissible fragment; the remaining obligations are enumerated explicitly in Section 13.

### 5.1 Metatheory: Adequacy of the Two-Layer System

The admissible fragment of this paper is paired with a typed bytecode, Op, defined in the companion paper "Op: A Typed Bytecode for Compliance-Carrying Operations." The central metatheoretic claim of the two-layer system is split across one theorem and two conjectures, with their respective scopes made explicit.

**Categorical substrate of the bridge.** Before stating the three claims it is useful to name the shared structure on which they rest. Lex and Op are two sectors of a single formal system, coupled at three points. First, they share the compliance-context monoid $(K, \bigwedge_K, \top_K)$ of §4.11 (meet-semilattice with top). The Lex-to-Op compilation $[\![\cdot]\!]$ is intended to be a strong monoidal functor on that monoid: $[\![\top_K]\!] = \top_K$ and $[\![\kappa_1 \bigwedge_K \kappa_2]\!] = [\![\kappa_1]\!] \bigwedge_K [\![\kappa_2]\!]$, which is the compatibility condition the companion paper states in its §8.2. Second, they share the verdict observable on the embedded Applicable fragment. Lex's `ComplianceVerdict` is a five-element bounded distributive Heyting algebra (Qed in `lex/formal/coq/Lex/VerdictHeyting.v`); Op carries concrete tensor cells with a compliance-grade axis plus applicability markers, and its Lex embedding obligations are meet-preservation on the shared Applicable verdict layer rather than an isomorphism of full mixed-axis tensors. Third, they share the PCAuth witness; a `PCAuth(auth, h, v, r)` authored under the Lex `discretion(auth)` effect (§3.5, §4.7) is re-verified independently at each receiving Op evaluator under the verifier obligations of the companion paper §8.3. Neither sector trusts the other's verifier; both re-derive the same verdict from the same content-addressed witness when the verifier obligations are discharged. The PCAuth witness is in this sense a shared certificate carrier in the proof-carrying-code discipline of Necula (1997): the intertwiner that lets the pure predicate layer (Lex) and the effectful operational layer (Op) agree on fill-site verdicts without either layer reaching into the other's internal state.

Let `L_adm` $\subseteq$ `Lex` be the admissible fragment defined above, and let `[[·]] : L_adm → Op` be the compilation given in the companion paper. The three claims are:

- **(a) Verdict soundness (target theorem; finite cases Qed-closed).** The target theorem is: for every `M ∈ L_adm` and every well-typed compliance context `ctx`, if Lex reduces `M` against `ctx` to a verdict `v`, then Op reduces the compiled bytecode against the compiled context to the same verdict `v`. The Qed-closed evidence currently covers the nine compilation cases recognised by `op/formal/coq/CompilationSoundness.v` (`verdict_preservation_*`). The universal theorem over every `M ∈ L_adm` remains open until the compiler coverage lemma maps every checker-accepted Lex constructor into those cases.

- **Completeness up to three Op-specific operational terminals (Conjecture).** For every `M ∈ L_adm` and `ctx`, if Op reduction of `[[M]]` against `[[ctx]]` terminates in a verdict `v`, then either Lex reduction of `M` against `ctx` also terminates in `v`, or the Op reduction terminated at one of three Op-specific operational terminals: gas exhaustion, callback timeout, or receiving-zone sanctions re-validation. The intended proof strategy is a coinductive bisimulation up-to-$\tau$ (Sangiorgi 1998): construct a bisimulation between Lex's reduction relation and Op's, with $\tau$-transitions standing in for Op's metering steps and zone-transit hops, and show that the only Op runs without a Lex preimage are those that hit one of the three terminals. The technical content is the same up-to-$\tau$ argument that Sangiorgi develops for CCS bisimulation, lifted to the Lex/Op pair. The present paper claims completeness as a conjecture, not as a theorem.
- **Full abstraction on compliance contexts (Conjecture).** For all `M, N ∈ L_adm`, `M` and `N` produce the same verdict in every Lex compliance context iff their compilations produce the same verdict in every Op compliance context. The intended proof strategy is a compliance-context simulation relation in the CompCert style (Leroy 2009): build a relation `R ⊆ L_adm × Op` that contains the pair `(M, [[M]])` for every well-typed `M`, is closed under reduction in both directions, and respects compliance-context observation. The technical content sits in showing that compiling distinct discretion-hole sites preserves observational distinctness, that pack-evolution rewrites do not glue distinct verdict trajectories under compilation, and that tribunal coercions compile to bytecode-level transport that an Op compliance-context observer can distinguish exactly when the Lex observer can. None of these three sub-arguments is reduced to the Lex/Op specifics yet; the present paper claims full abstraction as a conjecture with a proof strategy, not as a theorem.

The reproduction of the up-to-$\tau$ argument inline is left to a follow-on paper because it requires a careful case analysis over the entire Op operational semantics under each Lex constructor's compilation; that is a separate paper-length development. Until then, the central adequacy story has Qed-closed verdict preservation for nine compilation cases, an open coverage lemma for all of `L_adm`, plus two conjectural directions with fixed proof strategies (completeness modulo operational terminals; full abstraction on compliance contexts). The companion Op development also closes a deliberately finite verdict-agreement theorem in `LexOpAdequacy.v`, modulo `prelude` and `host_sanctions`; the descriptive theorem aliases are `lex_op_finite_verdict_agreement` and `lex_op_finite_verdict_agreement_bisim`. This is not Lex denotational adequacy or full abstraction. Modal operators, temporal coercions, pack evolution, unfilled holes, and the compliance-context full-abstraction theorem remain outside the Qed-closed finite-case soundness fragment.

The intent of the three-part adequacy claim is that Lex and Op are not two independent languages with a stapled compilation pass. They are two presentations of the same operational commitment: Lex specifies *what* a compliance rule decides; Op specifies *how* the same decision is computed, metered, and made replayable across sovereign zones. For the nine finite compilation cases covered by `verdict_preservation_*`, a developer authoring in Lex can transport settled verdict conclusions to the corresponding Op execution.

A replay operator verifying a cross-zone execution may cite the same finite-case theorem only for payloads whose compiled form is covered by those cases. The universal `L_adm` source-to-target theorem still requires the compiler coverage lemma, and the full bidirectional correspondence remains conjectural pending the up-to-tau bisimulation development and the compliance-context proof. Without that closure, the two layers stand connected one-way only on the Qed-closed finite source-to-target cases, with a separate finite Op-side verdict-agreement core; the general converse and the precise observational equivalence story are not yet proved.

The carve-out is minimal: three Op-specific terminals (gas exhaustion, callback timeout, sanctions re-validation at a receiving zone) have no Lex preimage because Lex does not express metering, callbacks, or multi-zone re-validation. The companion paper's *Carve-out remark* enumerates them explicitly.

**Admission carrier at the Lex -> Op boundary.** The public boundary object consumed by an admission host is an `AdmissionProgram` record:

```
AdmissionProgram = {
  op_payload_digest  : OpPayloadDigest,
  lex_source_digest  : LexSourceDigest,
  pack_digest        : PackDigest,
  context_digest     : ContextDigest,
  compiler_digest    : CompilerDigest,
  registry_digest    : PrimitiveRegistryDigest,
  gas_schedule_digest: GasScheduleDigest,
  oracle_log_digest  : OracleLogDigest,
  gas_bound          : Nat,
  effect_row         : EffectRow,
  required_receipts  : List LexReceipt,
  pcauth_entries     : List AcceptedFilledHole,
  failed_predicates  : List SafetyPredicate,
  deferred_predicates: List SafetyPredicate
}
```

The admission rule is fail-closed: the Op payload is admitted only if the payload, source, pack, context, compiler, primitive registry, gas schedule, PCAuth entries, oracle log digest, gas bound, and effect row agree with the receipt set and the observed environment. The Qed-closed receipt algebra proves locality for the receipt component: composing accepted receipts cannot hide unresolved obligations or discretionary frontiers. The public Rocq carrier `lex/formal/coq/Lex/AdmissionEnvelope.v` closes the structural admission facts: `admission_fails_closed_on_payload_mismatch`, `admission_fails_closed_on_unaccepted_receipts`, `admission_no_failed_or_deferred_predicates`, `admission_pcauth_entries_verified`, and `receipt_to_bundle_preservation`.

The executable Op checker now enforces the same narrow waist at the payload level: `let` initializers are checked against their annotations, non-unit declared outputs require a return, every `return` is checked against the program's declared output type, and expression-level `match` requires a `Bool` or finite-variant

scrutinee, rejects duplicate and unknown arms, requires full constructor coverage for known finite scrutinees, binds each arm payload at the constructor payload type, requires all arm result types to agree, and requires the materialized catch-all to have the same type. The Lex compiler's match and defeasible-rule lowering emits a typed unreachable catch-all, so an admissible Lex term cannot compile into an Op payload whose advertised output or branch result type is only paper-correct. What remains open is the full Op proof-bundle preservation theorem: constructing an executable Op proof bundle from the admitted envelope, proving that the bundle preserves the obligation frontier, PCAuth request bindings, gas/effect bounds, and context digest, and then proving that the Op verifier accepts exactly those bundles. This is the pragmatic narrow waist: Lex compiles down to an Op payload plus a proof-carrying admission envelope, and the envelope type-checks only when the orchestrated activity is precisely the one the Lex proof authorized.

The supporting mechanization remains:

- **Core scaffold.** `LexCore.v` formalizes the admissibility predicate, level resolution, hole authorization, tribunal coercion, temporal lift, summary compilation, and the derivation certificate. Both directions of admissibility decidability are proved at the scaffold level (the full-calculus decidability theorem is separately conditional on the remaining full-calculus metatheory, discussed above). Level non-self-application is proved (no `Lt L L` inhabitant). Hole authorization is proved in the forward direction. Temporal lift is total; temporal retract is not expressible. In `LexCore.v` the only classical dependencies exposed by `Print Assumptions` are `Classical_Prop.classic` and `Description.constructive_definite_description`, imported through Coq's `ClassicalDescription` library and used only by `principle_balancing_terminates`; a constructive Tarjan or Kosaraju decision procedure would replace that step without changing the theorem statement. In the wider `formal/coq/Lex/` tree, the mutually recursive syntax equality procedures `term_eq_dec`, `branch_eq_dec`, and `exception_eq_dec` in `Lex/Syntax.v` are now Qed-closed; `Print Assumptions` reports each closed under the global context. The remaining named non-constructive dependency disclosed here is `functional_extensionality_dep`, invoked by two lemmas in `Propagation/Graph.v` (`dm_le_antisym`, `evaluate_fixed`). It leaves the sixteen Qed-closed paper-level statements enumerated in the Mechanization Status appendix intact and appears under `Print Assumptions` for theorems that transit that call graph; it is therefore named rather than left implicit.
- **Flat admissible strong normalization.** `FlatAdmissibleSN.v` proves SN for the flat admissible fragment (variables, constants, accessors, let, match, defeasible with non-empty exception lists, and affine lambda) under call-by-value reduction. The proof is a well-founded induction on a size measure using an affine substitution lemma. `Print Assumptions flat_admissible_sn_ext` reports "Closed under the global context." The proof is fully constructive.
- **Temporal non-regression.** `lex/formal/coq/Lex/TemporalStratification.v` closes the object-language temporal invariant: no Time_0-introduction rule has a Time_1 premise, no Time_0 derivation contains a Time_1-to-Time_0 constructor step, and the generated temporal coercion graph

has no path from `Time_1` back to `Time_0`. `Print Assumptions temporal_non_regression` and `Print Assumptions no_temporal_retract` both report closed under the global context.

- **Pack re-evaluation soundness.** `lex/formal/coq/Lex/PackReevaluation.v` closes the re-evaluation proposition: derived legal time is a closure over `(Time_0 source, pack, pack-relative witness)`, and re-evaluation along a typed rewrite transports only the witness/pack component while preserving source-time history. The rewrite witness carries effective-date monotonicity and abstract conservativity/type-preservation fields. The theorems `re_evaluation_soundness`, `src_0_reevaluate`, `reevaluate_derivation_pack`, `reevaluate_id`, `reevaluate_compose`, `rewrite_effective_date_preserved`, `source_history_reevaluate_derived`, and `source_history_reevaluate_pack` are Qed-closed; `Print Assumptions re_evaluation_soundness` reports closed under the global context.
- **Effect monotonicity.** `lex/formal/coq/Lex/PaperMechanization.v` now defines a finite row-labelled `EffectDerivation` tree and closes `effect_monotonicity` by induction on the `subderivation` relation. A subderivation's row is always subsumed by the enclosing derivation row because parent rows are constructed by semilattice join. `Print Assumptions effect_monotonicity` reports closed under the global context.
- **PCAuth quorum extraction.** `lex/formal/coq/Lex/PCAuthQuorum.v` closes the structural verifier theorem `quorum_acceptance_unfolding`: an accepted quorum bundle exposes a finite attestation list whose signer identities are distinct, whose length meets the policy-required threshold, whose signers lie in the policy committee, and whose entries are valid for the exact signed payload at the admission snapshot. The supporting theorems `accepted_filled_hole_quorum`, `quorum_policy_binding`, `quorum_exact_payload`, `quorum_depth_bound`, `quorum_signature_payload_binding`, `quorum_anchor_chain_not_revoked`, and `quorum_admission_time_bound` expose the exact protocol tag, canonicalization version, hash/signature suite, authority, hole, value, request, mode, ledger, pack, context, committee, delegation-depth, signature, anchor, chain, and non-revocation components. `Print Assumptions quorum_acceptance_unfolding` reports closed under the global context.
- **Admission envelope.** `lex/formal/coq/Lex/AdmissionEnvelope.v` closes the structural fail-closed admission carrier: exact environment matching, receipt acceptance preservation, failed/deferred predicate rejection, and PCAuth-entry verification are Qed-closed. `receipt_to_bundle_preservation` proves that every receipt inside an admitted bundle is accepted.
- **Verdict Heyting algebra.** `lex/formal/coq/Lex/VerdictHeyting.v` mechanizes the five-element `ComplianceVerdict` bounded distributive Heyting algebra (thirteen laws: meet/join idempotence, commutativity, associativity, absorption, distributivity both directions, bounds,

and the Heyting identity `a ∧ (a → b) = a ∧ b`). Bundled into `verdict_is_heyting`, all Qed-closed by exhaustive case analysis on the finite five-element carrier.

- **Receipt algebra for admission hosts.** `lex/formal/coq/Lex/ReceiptAlgebra.v` defines a Lex receipt as `(verdict, obligations, unresolved, frontier)` and composes receipts by verdict meet and list concatenation on the observable obligations/frontiers, which is membership-equivalent to finite union. The Qed-closed theorems `accepted_compose_iff`, `compliant_compose_iff`, `composed_verdict_is_glb`, and `admission_locality` prove that no unresolved obligation or discretion frontier can be hidden by composition, and `Print Assumptions admission_locality` reports closed under the global context.
- **Bounded quantifiers.** `lex/formal/coq/Lex/BoundedQuantifiers.v` defines `bforall`, `bexists`, and `bexists_unique` over finite lists plus the iff-with-`In` correctness lemmas, distributivity under list concatenation, and the De Morgan duality `bforall_not_bexists`. A worked Seychelles IBC s.66 example closes `seychelles_s66_correct`. All Qed.
- **Defeasibility priority evaluator.** `lex/formal/coq/Lex/DefeasibilityPriority.v` mechanizes the priority-ordered evaluator `eval_defeasible` with soundness (`eval_sound`), stability, and override properties, plus the supporting lemmas `best_fired_in` (the selected exception is in the input list) and `best_fired_guard` (its guard fires). All Qed.
- **De Bruijn substitution metatheory.** `lex/formal/coq/Lex/DeBruijn.v` mechanizes shift and substitution over the full thirty-constructor Lex `Term` AST plus the companion `Branch` and `Exception` structures. Closed-under-substitution, `subst_above_free`, `subst_var_identity`, and the negative refutations of the naive unconditional shift-subst and subst-subst commutation are Qed-closed. The corrected well-scoped commutation `shift_subst_commute_ws` and substitution composition `subst_subst_ws` are Qed-closed by full mutual structural induction on `Term` / `Branch` / `Exception`. `Print Assumptions shift_subst_commute_ws` and `Print Assumptions subst_subst_ws` both report "Closed under the global context." This closes the substitution-metatheory obligation for binding forms.
- **Progress under operational step-extension.** `lex/formal/coq/Lex/Typing.v` extends the `step` inductive with eight new operational rules covering `Defeasible` and `Match` (scrutinee congruence, wildcard fire, constructor dispatch, head-shape skip, plus Defeasible empty/peel) and extends `value` with five new constructors (literals, axiom uses, constructor-headed inductive-intro values). Progress is Qed-closed over the full Lex `Term` AST without fragment restriction, conditional on two named Prop premises: `confluence_property` (used in the `T_App` and `T_Conv` canonical-forms arguments; itself reduces to the open `par_diamond_spec` in `Confluence.v`) and `match_exhaustiveness_property` (used in the `T_Match` canonical-forms argument to rule out the case where a scrutinee in WHNF matches no branch). The match premise has now been sharpened: `branches_cover_value` defines branch-list coverage, `match_coverage_property` requires every value reduct of the scrutinee to be covered, and Rocq

closes `match_coverage_implies_exhaustiveness` plus `progress_from_match_coverage : confluence_property → match_coverage_property → progress_property`. Confluence remains open at the diamond-lemma level; the remaining match work is to have the checker or `T_Match` construct `match_coverage_property` from finite constructor coverage. The companion skeleton `Typing_progress_skeleton.v` has been cleaned of its two obsolete `P→P` premise-lemma placeholders for `Defeasible` and `Match`; those cases are now carried directly by the operational step rules.

- **Tensor-alignment with the neighbouring compliance algebra.** `lex/formal/coq/Lex/TensorAlignment.v` mechanises the projection `lex_to_tensor` from the Lex per-coordinate five-chain to the tensor-factor type (three-chain compliance grade × orthogonal applicability marker). The projection is injective (`lex_to_tensor_injective`); on the Applicable fragment, the Lex chain meet commutes with the tensor meet (`lex_meet_preserves_applicable_fragment`, `lex_tensor_diagram_commutes_on_applicable`); outside the Applicable fragment, two Qed-closed counterwitnesses (`lex_meet_loses_provenance_on_mixed_axis`, `lex_meet_loses_provenance_exempt_vs_pending`) mechanise one direction of the mixed-axis impossibility dichotomy. The per-coordinate five-chain and the tensor-factor algebra are therefore not in conflict: they are compositional-level views, meet-preserving in projection on their shared sub-lattice and separated by provenance loss outside it.
- **Propagation-graph alignment.** `lex/formal/coq/Lex/PropagationAlignment.v` mechanises the isomorphism between Lex's `verdict` (five-element Heyting chain) and `Propagation.Graph.ComplianceState` (the propagation-graph monotonicity layer): `verdict_to_cs` / `cs_to_verdict` are Qed-closed mutual inverses (`verdict_cs_iso_left`, `verdict_cs_iso_right`); ranks are preserved (`verdict_to_cs_preserves_rank`, `cs_to_verdict_preserves_rank`); Heyting meet and propagation meet commute with the morphism (`verdict_to_cs_preserves_meet`, `cs_to_verdict_preserves_meet`); ordering relations agree (`verdict_to_cs_preserves_order`, `cs_to_verdict_preserves_order`). Every theorem Qed-closed at one layer transports to the other through the isomorphism.
- **Step-relation repair and parallel-reduction scaffolding.** `lex/formal/coq/Lex/Typing.v` carries forty-five binder-congruence step rules, one per reducible subterm per binder-bearing constructor (Lambda, Pi, Sigma, Rec, Let, Annot, Match, Defeasible, Pair, Proj, SanctionsDominance, DefeatElim, Lift0, Derive1, ModalAt / Eventually / Always / Intro / Elim, Hole, HoleFill, PrincipleBalance, Unlock, InductiveIntro args), closing the non-confluence gap identified (a `step_app_arg` with no congruence under binders produced a concrete Qed-witnessed non-joinable App / Lambda divergence). The head-step relation is preserved as `head_step` (twelve rules covering beta, zeta, annotation erasure, and the operational Defeasible / Match rules); `value_no_head_step` is Qed-closed against the non-congruence variant so the progress-theoretic notion of a value stays intact. Shape-preservation machinery is Qed-closed for every outer constructor (atomic-is-irreducible lemmas for TSort / Var / Constant / IntLit / RatLit /

StringLit / AxiomUse; Pi / Lambda / InductiveIntro outer-constructor preservation; the lifted `steps_*` variants), plus `step_preserves_value` and `steps_preserves_value` via strong induction on term size. `conv_eq_of_values_outer_eq` is the full ten-case conv_eq disjointness lemma. The parallel-reduction infrastructure lives in `lex/formal/coq/Lex/Confluence.v`: a `par` inductive with forty-six constructors (eight atomic, twenty-seven congruence, nine head-redex-fire, two Branch / Exception mutual forms), plus `par_branch` / `par_exception` mutual inductives; `par_refl` Qed over Term / Branch / Exception (with `Forall2_par_refl` / `_branch_refl` / `_exception_refl` helpers); thirty-three per-constructor `steps_X` congruence lemmas Qed; the two-direction bridge `step_implies_par` (Qed, fifty-six cases) and `par_implies_steps` (Qed, forty cases) connecting the single-step and parallel-step theories; and the bridge theorem `confluence_from_par_star_diamond : par_star_diamond_spec → confluence_spec` (Qed). The remaining Tait-Martin-Löf obligations (`par_subst_spec`, `par_shift_spec`, `par_diamond_spec`, `par_star_diamond_spec`) are retained as named Prop specifications (no `Admitted`, no `Axiom`, no `Hypothesis`) for the follow-on diamond-lemma development.

- **Typing-rule binder well-typedness and `has_type_well_scoped`.** `lex/formal/coq/Lex/Typing.v` typing rules for binder-bearing forms now carry the well-typedness premises standard for dependent type theories: `T_Lambda` adds `has_type ctx A (type_level i)` on the domain, `T_Let` adds the analogous premise for the bound type, `T_Defeasible` replaces a placeholder with two `Forall` premises (exception guards at `prop`, bodies at `base_ty`) using projection functions `exn_guard` / `exn_body` to satisfy strict positivity, and `T_Match` adds a `binder_tys_list` witness plus two `Forall2` premises (arity-match with each branch's pattern, each branch body typed at `return_ty` in the context extended by binder types) via projections `branch_pat` / `branch_body` / `pattern_arity`. `has_type_well_scoped : forall ctx t T, has_type ctx t T → well_scoped (length ctx) t` is Qed-closed by Gallina `Fixpoint` structural recursion on the typing derivation, with two nested `fix` helpers threading through the `Forall` / `Forall2` premises for T_Defeasible and T_Match. Supporting Qed results: `well_scoped_mono` (mutual), `ctx_lookup_lt_length`, `subst_preserves_ws` (mutual), `subst_shift_01_identity` (mutual), the three `ws_list_*_iff_Forall` bridges between the internal `fix` shape and the `Forall`-predicate shape, and the `well_scoped_*_from_Forall` companions. Print Assumptions reports "Closed under the global context" for each.
- **Companion Op verdict preservation.** The separate Lex→Op preservation development at `op/formal/coq/CompilationSoundness.v`, not the Lex core scaffold, carries two `Parameter`s (`host_sanctions` and `prelude`) representing external host primitives bound at deployment time. The two-valued range obligation on `host_sanctions` is not a file-level `Axiom`: it is a `Hypothesis` inside `Section SanctionsRangeSanity`, scoped to the two-valued sanity Example so any consumer discharges it with a proof for the concrete host they are binding. The main

preservation theorem `verdict_preservation_sanctions` depends only on the deterministic `host_sanctions` parameter itself and carries no range obligation.

- **Weakening (generalized insert-at-k form).** `lex/formal/coq/Lex/Typing.v` and `DeBruijn.v` mechanize an eager-shift context representation in which `ctx_extend ctx A := shift 0 1 A :: shift_ctx 0 1 ctx`, matching the standard CC/PTS presentation (MetaCoq's `rel_context` with `lift_rel_context` on insertion). Supporting `shift_ctx` and `insert_at` infrastructure is Qed-closed: `shift_ctx_length`, `shift_ctx_nth_error`, `shift_ctx_app` / `_firstn` / `_skipn`, the context-level swap `shift_ctx_shift_ctx_swap_0_1`, and the positional-insertion primitive `insert_at k A ctx` with its five lookup lemmas (`insert_at_0`, `insert_at_length`, `insert_at_lookup_lt` / `_eq` / `_gt`) and the structural identity `insert_at_ctx_extend_commute` consumed by T_Pi / T_Lambda / T_Let. A new DeBruijn commutation `shift_subst_commute_above : forall t s i c d, i ⩽ c → shift c d (subst i s t) = subst i (shift c d s) (shift (S c) d t)` is Qed-closed by full mutual structural induction on `Term` / `Branch` / `Exception`; it is the `c ⩾ i` complement of the existing `shift_subst_commute_ws`, and is exactly what the T_App and T_Let cases of weakening require. The main theorem `weakening_at_fix : conv_eq_shift_compat_spec → weakening_at_property` is Qed-closed as a Gallina `Fixpoint` on the typing derivation. Thirteen of the fourteen typing-rule cases are dispatched directly (T_Var via `insert_at_lookup_lt` / `_gt`; the five sort and time cases via `shift_sort` / `shift_type_level`; T_Pi / T_Lambda / T_Let via `insert_at_ctx_extend_commute` applied to the IH at depth `S k`; T_App / T_Let substitutional forms via `shift_subst_commute_above`; T_Annot and T_Conv directly; T_Defeasible via two inner fixes on the `Forall` premises over exception guards and bodies). The fourteenth case, T_Match, is dispatched inside the same conditional proof by a dedicated inner fix that consumes the IH-shifted scrutinee, return type, and per-branch body typings under the new arity-shifted body premise; `weakening_at_match_holds : conv_eq_shift_compat_spec → weakening_at_match_spec` shows that no additional match-specific premise remains. The T_Defeasible inner fixes walk the `Forall` structurally, calling the outer Fixpoint on each head `has_type` sub-derivation; Coq accepts the recursion because each head proof is structurally smaller than the enclosing `Forall`, which is itself a direct argument of the outer `T_Defeasible` constructor. The corollary `weakening_fix_conditional` reduces the depth-indexed theorem to `k = 0` via `insert_at_0`, giving `weakening_property` conditional on exactly one Prop spec, `conv_eq_shift_compat_spec`. Supporting shift-compatibility lemmas `value_shift_compat` (Fixpoint over the value derivation with an inner fix on `Forall value`), `shift_args_at` with its layered-cutoff structure, `shift_subst_args_commute`, and `branch_head_matches_shift` are Qed-closed and support the future closure of `conv_eq_shift_compat_spec` through `step_shift_compat` / `steps_shift_compat`. Two structural facts are load-bearing. (1) A body premise that types each branch body at `return_ty` without a pattern-arity shift is not stable under weakening: at depth `k` inside a branch's extended context it produces `shift (k + arity)`

`1 return_ty` where T_Match's weakened conclusion wants `shift k 1 return_ty`. The two agree only when `return_ty` has no free variables in `[k, k + arity - 1]`, which the typing rule does not enforce. The body premise therefore uses `shift 0 arity return_ty`, which preserves semantics and canonicalises the shift behaviour; the inner fix closes the match case under the same `conv_eq_shift_compat_spec` premise with `Qed`. (2) The companion operational rule `step_match_ctor_fire` carries a deeper structural defect that a layered pre-shift of constructor arguments does not address. Pre-shifting `args` via `shift_args_at 0 n args` before the substitution does not achieve shift-equivariance, because `subst_args` is a fold of single-index substitutions whose cutoff varies with the position of each argument in the fold; pre-shifting at a single uniform cutoff cannot match the position-dependent shifts that outer-context shifting produces. A concrete counter-witness is `args = [v_0; v_5]`, `body = v_0`, outer cutoff `c = 0`, shift amount `d = 1`: the layered reduct yields `v_6` while shift-then-fire yields `v_0`. The genuine resolution requires a new parallel multi-substitution primitive `parallel_subst : list Term → Term → Term` in the DeBruijn library with the appropriate shift-commutation lemmas, and a rewrite of `step_match_ctor_fire`'s reduct to use that primitive. That work is open and is the gating obligation for `conv_eq_shift_compat_spec`. The substitution-metatheory obligation for the standard binding forms is not among the open questions: it is discharged by `shift_subst_commute_ws` / `subst_subst_ws` / `shift_subst_commute_above`.

No additional open targets remain inside `PaperMechanization.v` itself. Broader core metatheory still in active work: preservation and progress for the full admissible fragment with Pi-types, effect rows, and discretion holes (the progress theorem is Qed-closed *conditional on* `confluence_property` and `match_exhaustiveness_property`, and the stronger certificate route `confluence_property → match_coverage_property → progress_property` is Qed-closed as `progress_from_match_coverage`; the unconditional form still depends on confluence and construction of the coverage certificate); strong normalization for the admissible fragment with dependent matches and modals; and the full-calculus extension of the administrative WHNF bound. Confluence is open: `par_diamond_spec` is a named Prop, not a Qed, so any claim that the calculus is confluent is conditional on its closure. Preservation for the full admissible fragment is not Qed-closed and is not separately claimed in this paper. Weakening for the full calculus is Qed-closed conditionally on one residual Prop spec, `conv_eq_shift_compat_spec`; `weakening_at_match_spec` is discharged by `weakening_at_match_holds` only under that same premise, so there is no separate match-specific weakening premise. Discharge of `conv_eq_shift_compat_spec` is gated by the open `step_match_ctor_fire` resolution: a parallel multi-substitution primitive in the DeBruijn library, replacing the position-dependent `subst_args` fold whose interaction with outer-context shifts is structurally incompatible with shift-equivariance under any uniform pre-shift. This obligation is acknowledged rather than claimed.

**Mechanization Status (per-statement enumeration)** This appendix enumerates every theorem, proposition, lemma, and conjecture stated in the body of this paper, in order of first appearance, and classifies each as one of four statuses:

- **Qed-closed (Q):** A Rocq proof closed with `Qed`. The Rocq source file is named.
- **Conditional (C):** A Rocq proof closed with `Qed` *conditional on* one or more named Prop specs whose discharge is open. The conditioning specs are named.
- **Open (O):** A named open `Prop` specification, a paper-only proof sketch with no Rocq target, or a statement whose sketch does not constitute a complete proof.
- **Conjecture (J):** Self-declared as a conjecture in the body, not as a theorem.

The tally below uses the convention: percentage = Qed-closed / (total minus Conjectures), since conjectures self-declare as open and do not belong in the denominator of a "what fraction is mechanized" question. Counted by this convention: 16 Qed-closed of 27 non-conjectural ledger entries = 59.3%. If the strict convention is used (percentage = Qed-closed / total): 16 of 30 = 53.3%. Entries that are support kernels for stronger obligations are marked as such in the note column. Cut elimination is reframed as a conjecture in the body (entry #30 below) and is therefore in the conjecture column rather than the open column; the resulting twenty-seven-or-thirty split below holds with cut elimination counted as a conjecture.

| # | Statement | Section | Status | Rocq target / note |
|---|---|---|---|---|
| 1 | Re-evaluation soundness | 3.3 | Q | `PackReevaluation.v` (`re_evaluation_soundness`, `src_0_reevaluate`, `reevaluate_compose`; `Print Assumptions re_evaluation_soundness` closed under the global context) |
| 2 | Pack-evolution soundness, stable fragment | 3.3 | O | Paper-only (proof sketch by induction on derivations) |
| 3 | Pack-evolution soundness, modal/temporal subsystem | 3.3 | J | Self-declared conjecture |

| # | Statement | Section | Status | Rocq target / note |
|---|---|---|---|---|
| 4 | Temporal rule-graph non-regression | 3.2 | Q | `TemporalStratification.v` (`no_time1_premise_to_time0_rule`, `temporal_non_regression`, `no_temporal_retract`; `Print Assumptions temporal_non_regression` closed under the global context) |
| 5 | Finite observation event-union support for discretion-hole reductions | 3.5 | Q | `finite_observation_event_union_bound` in `PaperMechanization.v` closes the finite-observation counting lemma; full cryptographic instantiation remains an open verifier/reduction obligation |
| 6 | Quorum acceptance unfolding | 3.5 | Q | `PCAuthQuorum.v` (`quorum_acceptance_unfolding`, `accepted_filled_hole_quorum`, `quorum_policy_binding`, with policy threshold/committee binding, exact signed-payload, request/pack/context, depth-bound, signature-binding, anchor/chain/non-revocation and admission-time support; `Print Assumptions quorum_acceptance_unfolding` closed under the global context) |
| 7 | PCAuth forgery reduction | 3.5 | O | Paper-only EUF-CMA reduction with bounded quorum-width slot-guessing and quorum-position hybrid argument; not mechanized |
| 8 | Prop erasure | 4.1 | O | Paper-only proof sketch by induction on typing derivation |

| # | Statement | Section | Status | Rocq target / note |
|---|---|---|---|---|
| 9 | Finite-collection quantifiers admissible | 4.2 | Q | `BoundedQuantifiers.v` closes finite-list boolean laws; `BoundedQuantifierAdmissibility.v` closes admissible elaboration to finite boolean folds (`bounded_forall_term_admissible`, `bounded_exists_term_admissible`) |
| 10 | Bounded Skolemization | 4.2 | O | Paper-only proof using `choose` over a finite enumeration; not mechanized |
| 11 | Effect monotonicity | 4.3 | Q | `effect_monotonicity` in `PaperMechanization.v`; `Print Assumptions effect_monotonicity` closed under the global context |
| 12 | Defeasible effect join | 4.4 | Q | `defeasible_effect_join` in `PaperMechanization.v` |
| 13 | Non-accidental-laundering soundness | 4.6 | O | Paper-only proof by induction on typing derivation |
| 14 | Bridge 2-cell congruence | 4.6 | Q | `BridgeSemantics.v` (`function_bridge_eq_equivalence`, `function_bridge_whiskering_laws`, `function_modal_action_respects_2cells`) |
| 15 | Canonical bridge bicategory / strict function target | 4.6 | Q | `BridgeSemantics.v` (`function_bridge_bicategory_laws`) closes the strict function-bridge target; proof-relevant raw syntax quotienting remains part of adequacy |
| 16 | Tribunal modal as canonical bridge action / 2-functor target | 4.6 | Q | `BridgeSemantics.v` (`function_bridge_modal_action_laws`, `function_bridge_strict_2functor_coherence`) |
| 17 | Admissibility lift for authority recognition | 4.6 | O | Paper-only proof via finite-fixpoint saturation; not mechanized |

| # | Statement | Section | Status | Rocq target / note |
|---|---|---|---|---|
| 18 | Delegation does not expand authority | 4.7 | O | Paper-only proof sketch by induction on the delegation derivation |
| 19 | Delegation soundness | 4.8 | O | Paper-only proof by structural induction on the delegation chain |
| 20 | Revocation determinism | 4.9 | O | Paper-only proof by trichotomy on bulletin-ordered timestamps |
| 21 | Administrative WHNF-bounded reduction | 5 | Q | `administrative_whnf_bounded_reduction` in `PaperMechanization.v` closes the typed administrative weak-head fragment by `whnf_head_steps`; the full-calculus extension remains under the metatheory frontier |
| 22 | SN-admissible | 5 | J | Self-declared conjecture; flat fragment Qed in `FlatAdmissibleSN.v`, full admissible fragment open |
| 23 | Verdict lattice laws | 7.4 | Q | `VerdictHeyting.v` (idempotence, commutativity, associativity, absorption, distributivity, boundedness; thirteen laws bundled into `verdict_is_heyting`) |
| 24 | ComplianceVerdict is a Heyting algebra | 7.4 | Q | `verdict_is_heyting` in `VerdictHeyting.v` |
| 25 | Receipt locality / no-laundering | 7.4 | Q | `ReceiptAlgebra.v` (`accepted_compose_iff`, `compliant_compose_iff`, `admission_locality`; `Print Assumptions admission_locality` closed under the global context) |
| 26 | Admission envelope fail-closed / receipt preservation | 5.1 | Q | `AdmissionEnvelope.v` (`admission_fails_closed_on_payload_mismatch`, `admission_fails_closed_on_unaccepted_receip`, `admission_no_failed_or_deferred_predicates`, `admission_pcauth_entries_verified`, `receipt_to_bundle_preservation`) |

| # | Statement | Section | Status | Rocq target / note |
|---|---|---|---|---|
| 27 | Priority evaluator agreement | 7.4 | Q | `DefeasibilityPriority.v` (`eval_defeasible_normalized_meet` proves equality with the normalized meet form under strictly descending priorities; `eval_sound` remains the support lemma) |
| 28 | Pack-rewrite preservation | 7.4 | O | Paper-only proof sketch; relies on append-only re-evaluation semantics |
| 29 | Adequacy of the presheaf model | 10 | O | Sketch with explicit case enumeration: standard CwF cases (variable, sort, dependent function, lambda, application, let), plus six Lex-specific cases (dependent match, defeasibility, tribunal bridge transport, priority-graph defeasibility, temporal stratification, typed discretion holes) each requiring its own non-CwF argument; mechanization open beyond the flat fragment |
| 30 | Cut elimination for the admissible fragment | 10 | J | Reframed in the body from Theorem to Conjecture; conditional on the open subject reduction conjecture and the open SN-admissible conjecture |

Tally: 16 Qed-closed (Q), 0 Conditional (C), 11 Open (O), 3 Conjecture (J), total 30. Non-conjectural denominator = 30 − 3 = 27. Qed-closed fraction = 16 / 27 = 59.3%.

Op-paper soundness (direction (a) of the §5.1 adequacy claim, mechanized in `op/formal/coq/CompilationSoundness.v` across nine `verdict_preservation_*` cases) is *not* a Lex paper-level statement and is therefore not in the table; it appears as the source for the §5.1 narrative claim that direction (a) is Qed-closed.

Supporting mechanization items (`weakening_at_fix`, `shift_subst_commute_ws`, `shift_subst_commute_above`, `subst_subst_ws`, `has_type_well_scoped`, the `par`-relation infrastructure, `verdict_to_cs` / `cs_to_verdict` mutual inverses, `lex_to_tensor` injectivity, `step_implies_par` and `par_implies_steps` bridges) underwrite the paper-level statements but are not themselves paper-

level statements; they are described in the supporting-mechanization bullets immediately above this appendix.

The deeper structural finding on the operational rule `step_match_ctor_fire` (resolution open, gating `conv_eq_shift_compat_spec`) is also disclosed at the rule's companion typing rule in §4.2. The original T_Match body-premise pattern-arity gap, also disclosed in §4.2, has now been repaired by the arity-shift refactor; the inner-fix `weakening_at_match_holds` closes the match weakening case under the same `conv_eq_shift_compat_spec` premise as the rest of weakening.

---

## 6. The Prelude

The admissible fragment operates within a compliance prelude: a global signature providing the type vocabulary for jurisdictional rules. The prelude is organized into:

- **Core types**: `IncorporationContext`, `ComplianceVerdict`, `SanctionsResult`, `Bool`, `Nat`, `ComplianceTag`, and finite collection families such as `Collection(T)`, `List(T)`, `Set(T)`, and `Multiset(T)`.
- **Verdict constructors**: `NonCompliant`, `Pending`, `NotApplicable`, `Exempt`, `Compliant`. `ComplianceVerdict` is a bounded five-element chain ordered by legal restrictiveness, with the order, the meet/join semantics, and the Heyting structure proved in Section 7.4.
- **Boolean constructors**: `True`, `False`.
- **Sanctions constructors**: `Clear` plus the `sanctions_query` effect.
- **Tag constructors and accessors**: jurisdiction-specific enumerated values (e.g., `FullLicense`, `InPrincipleApproval`, `FitAndProperSatisfied`) together with the field projections that extract them from an `IncorporationContext` (e.g., `fsra_authorization_status`, `director_count`).
- **Collection combinators**: enumeration, membership proofs, `filter`, and cardinality `|C|` for finite-collection quantification and threshold rules.
- **Boolean, numeric, and sanctions accessors**, and explicit combinators for threshold and predicate operations.

All prelude types live at universe level 0. The prelude is the vocabulary within which the admissible fragment operates; it is versioned per release of the logic. A rule authored against one prelude version is portable across implementations that agree on that version.

---

# 7. Worked Examples

The examples span both the admissible fragment and the full calculus, and are mechanized in `lex/formal/coq/Lex/Examples/*.v`. Sections 7.1 and 7.4 are fully admissible: they use only flat pattern matching over prelude types, let, lambda, and defeasible forms. The remaining sections exercise features outside the admissible fragment: Section 7.2 uses dependent pattern matching and a discretion hole; Section 7.3 exercises the `sanctions_query` effect; Section 7.5 traces a discretion-hole lifecycle; Section 7.6 combines sanctions, typed discretion holes, tribunal modals, and a derived reporting deadline; Section 7.7 combines defeasibility, temporal stratification, and a discretion hole; Section 7.8 uses the temporal coercion `derive_1` with stacked `Toll` rewrites. The admissible-fragment examples compile and typecheck in the current implementation; the richer examples are expressible in the core calculus and illustrate the features the admissible fragment targets once the stratification extensions of Section 13 are integrated.

## 7.1 Defeasible Rule: BVI Director Residency

A rule encoding the British Virgin Islands Business Companies Act 2004, s.111 requirement that a company must have at least one director. The section fixes the statutory minimum; s.31 governs a distinct private-company classification whose filing and governance surface is narrower than the public-company baseline that the minimum-director rule otherwise presumes:

```
-- Rule: Director minimum (BC Act 2004, s.111)
defeasible
  lambda (ctx : IncorporationContext).
    match ctx.director_count return ComplianceVerdict with
    | Zero => NonCompliant
    | _ => Compliant
  priority 0
  unless
    lambda (ctx : IncorporationContext).
      match ctx.private_company_s31_classification return Bool with
      | True => True
      | _ => False
    priority 1
end
```

**Typing derivation (informal).** The outer term is a `Defeasible` node. The base type is `Pi(ctx : IncorporationContext). ComplianceVerdict`. The base body matches on `director_count` (a `Nat` accessor): zero directors yields `NonCompliant`, any positive number yields `Compliant`. The exception at priority 1 checks whether the entity carries the BC Act s.31 private-company classification, under which the filing and governance surface narrows relative to the public-company baseline. The type checker

verifies: (1) the base body inhabits the base type, (2) `director_count` is a valid `Nat` accessor in the prelude, (3) the exception guard returns `Bool`, (4) all constructors (`Zero`, `True`, `False`, `Compliant`, `NonCompliant`) are in the prelude.

This differs from the ADGM rule in Section 3.1 in its use of a numeric accessor (`director_count`) and its demonstration that defeasible exceptions can exempt entities from requirements entirely, including cases beyond severity modulation.

### 7.2 Cross-Jurisdictional Rule with Dependent Types

The admissible fragment handles flat pattern matching over prelude types. The richer dependent type machinery is designed for rules that the admissible fragment cannot express. A rule requiring dependent types, beyond the admissible fragment but expressible in the full calculus:

```
lambda (j : Jurisdiction).
  lambda (ctx : EntityContext j).
    match j return (Pi(_ : EntityContext j). ComplianceVerdict) with
    | ADGM => lambda (c : EntityContext ADGM).
        match c.fit_and_proper_status return ComplianceVerdict with
        | FitAndProperSatisfied => Compliant
        | FitAndProperUnderReview => Pending
        | _ => ?fp_adgm : ComplianceVerdict
               @ authority ADGM.FSRA
               scope { jurisdiction: ADGM }
    | Seychelles => lambda (c : EntityContext Seychelles).
        match c.director_resident return ComplianceVerdict with
        | True => Compliant
        | False => NonCompliant
```

Here, `EntityContext` is a type family indexed by jurisdiction. The context for an ADGM entity carries fields specific to ADGM (FSRA authorization status, fit-and-proper assessments). The context for a Seychelles entity carries fields specific to Seychelles (director residency). The return type of the outer match depends on the jurisdiction, it is `Pi(_ : EntityContext j). ComplianceVerdict` where `j` is bound by the outer lambda.

The ADGM branch contains a typed discretion hole `?fp_adgm`: when the fit-and-proper status is neither satisfied nor under review, the machine cannot determine compliance. A judgment of type `ComplianceVerdict` must be supplied by the ADGM FSRA. The Seychelles branch has no discretion hole, director residency is a binary fact.

This rule cannot be expressed in Catala (no dependent types, no discretion holes), in L4 (no jurisdiction-indexed type families), or in any conventional language without losing the typing guarantees. The depen-

dent type ensures that an ADGM-specific accessor is never applied to a Seychelles context, and vice versa. The discretion hole ensures that the "fit and proper" determination is not silently approximated.

### 7.3 Sanctions Hard-Block

A rule under the Seychelles Anti-Money Laundering and Countering the Financing of Terrorism Act 2020, which discharges the sanctions-screening obligations imposed on reporting entities by the Financial Intelligence Unit and by UN-derived designations incorporated into Seychelles law:

```
lambda (ctx : IncorporationContext) [sanctions_query].
  let incorporator_clear : SanctionsResult = sanctions_check ctx.incorporator in
  let directors_clear : SanctionsResult = sanctions_check_all ctx.directors in
  let owners_clear : SanctionsResult = sanctions_check_all ctx.beneficial_owners in
  match (incorporator_clear, directors_clear, owners_clear)
    return ComplianceVerdict with
  | (Clear, Clear, Clear) => Compliant
  | _ => NonCompliant
```

Note the effect annotation `[sanctions_query]` on the lambda. This rule exercises the distinguished sanctions effect. It is *not* a defeasible rule, there is no `defeasible ... unless ... end` wrapper. Sanctions screening is non-defeasible by design. Lawful licenses, exemptions, delisting evidence, or shared-authority recognition enter before this rule as sanctions facts or rule-pack witnesses. Once the rule derives sanctions non-compliance for the applicable scope, the result is a hard block that no ordinary defeasible exception overrides.

The type checker tracks the `sanctions_query` effect. Any function calling this rule must declare the sanctions effect in its own effect row, or the effect subsumption check fails. This is a compile-time guarantee that sanctions checks are never silently elided.

### 7.4 Verdict Lattice

In practice, an entity is subject to multiple compliance rules from multiple statutes in the same jurisdiction. A **fiber** is a single typed compliance evaluation: one rule applied to one entity producing one `ComplianceVerdict` with an associated compliance domain. The term is load-bearing. Let `D` be the finite set of compliance domains (the 23-domain vocabulary of the compliance-tensor companion note) and let `V` be the five-element verdict chain fixed below. The Lex compliance structure is a constant Grothendieck fibration `p : V x D → D` in the product sense (Grothendieck 1971; cf. Johnstone, *Sketches of an Elephant*, 2002, B1.3): the domain of discourse is the trivial fibration whose fiber over each domain `d in D` is the same verdict lattice `V`. A single rule applied to an entity produces a section of this fibration on the singleton domain assigned by the rule; a full evaluation of an entity against a rule pack produces a partial section `s : D' → V` for `D'` the subset of domains touched by the pack. Composition inside one domain is the fibrewise Heyting meet defined below; aggregation across domains lives one layer above, in

the compliance tensor algebra of the companion note. An **evaluation engine** consumes fibers produced by rules applied to entities and composes their results fibrewise. A **pack** is a versioned bundle of compliance rules for a jurisdiction (for example, all rules from the Seychelles IBC Act 2016), organized by legal basis and compliance domain; a pack selects which sections of the fibration are live at a given event time.

The carrier is the bounded five-element set

$$\mathcal{V} = \{\texttt{NonCompliant}, \texttt{Pending}, \texttt{NotApplicable}, \texttt{Exempt}, \texttt{Compliant}\}$$

with the total order

$$\texttt{NonCompliant} < \texttt{Pending} < \texttt{NotApplicable} < \texttt{Exempt} < \texttt{Compliant}.$$

The order is by legal permissiveness, equivalently by inverse restrictiveness:

- `NonCompliant`: a hard fail.
- `Pending`: the rule is live but a required check has not yet been disposed.
- `NotApplicable`: the rule does not reach this entity.
- `Exempt`: the rule reaches the entity but a statutory exemption defeats the obligation.
- `Compliant`: the rule applies and is satisfied.

The passing states are exactly those strictly above `Pending`: `{NotApplicable, Exempt, Compliant}`. This is a single order of legal restrictiveness. Belnap's four-valued logic (1977) is the nearby contrast point: it separates truth and information into two orders. Lex does not need a bilattice here. It needs one monotone axis that distinguishes "the rule never attached" from "the rule attached and was lifted by an exemption."

Lex composes verdicts with the lattice operations on this chain:

```
verdict_meet(a, b) = min(a, b)   -- restrictive composition
verdict_join(a, b) = max(a, b)   -- permissive composition
```

`verdict_meet` is used for conjunctive composition: every fiber in a domain must clear the bar, so the most restrictive verdict wins. `verdict_join` is used for disjunctive composition: if any one of a family of sub-rules independently licenses the action, the least restrictive successful verdict wins. The lattice facts used below are standard finite-chain facts in the sense of Birkhoff's *Lattice Theory*; the Heyting-algebra reading is the standard locale-theoretic one in Johnstone's *Stone Spaces*.

**Proposition (verdict lattice laws).** For all $a, b, c \in \mathcal{V}$:

1. Idempotence: $a \wedge a = a$ and $a \vee a = a$.
2. Commutativity: $a \wedge b = b \wedge a$ and $a \vee b = b \vee a$.
3. Associativity: $(a \wedge b) \wedge c = a \wedge (b \wedge c)$ and $(a \vee b) \vee c = a \vee (b \vee c)$.

4. Absorption: $a \wedge (a \vee b) = a$ and $a \vee (a \wedge b) = a$.
5. Distributivity: $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ and $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$.
6. Boundedness: $\bot =$ `NonCompliant` and $\top =$ `Compliant`, so $\bot \wedge a = \bot$ and $\top \vee a = \top$.

*Proof sketch.* Because $\mathcal{V}$ is a total order and $\wedge = \min$, $\vee = \max$, idempotence, commutativity, associativity, absorption, and boundedness are immediate. For distributivity, assume without loss of generality that $b \leq c$. Then $b \vee c = c$ and $b \wedge c = b$. Also $a \wedge b \leq a \wedge c$ and $a \vee b \leq a \vee c$, so

$$a \wedge (b \vee c) = a \wedge c = (a \wedge b) \vee (a \wedge c)$$

and

$$a \vee (b \wedge c) = a \vee b = (a \vee b) \wedge (a \vee c).$$

The dual case $c \leq b$ is symmetric.

**Theorem (ComplianceVerdict is a Heyting algebra).** Define implication on $\mathcal{V}$ by

$$a \Rightarrow b = \begin{cases} \top & \text{if } a \leq b, \\ b & \text{if } b < a. \end{cases}$$

Then for every $x, a, b \in \mathcal{V}$,

$$x \leq (a \Rightarrow b) \quad \iff \quad x \wedge a \leq b.$$

Hence $\mathcal{V}$ is a bounded distributive lattice with relative pseudocomplements, therefore a Heyting algebra. In particular,

$$a \wedge (a \Rightarrow b) = a \wedge b.$$

The pseudocomplement is $\neg a := (a \Rightarrow \bot)$.

*Proof sketch.* If $a \leq b$, then $a \Rightarrow b = \top$. The left side is always true, and the right side holds because $x \wedge a \leq a \leq b$. If $b < a$, then $a \Rightarrow b = b$. Since $\mathcal{V}$ is totally ordered, $x \wedge a \leq b$ holds exactly when $x \leq b$, which is exactly the left side. This is the residuation law. The preceding proposition supplies bounded distributivity, so the relative pseudocomplements make $\mathcal{V}$ a Heyting algebra. The identity $a \wedge (a \Rightarrow b) = a \wedge b$ is the same two-case analysis: if $a \leq b$, both sides are $a$; if $b < a$, both sides are $b$.

The Seychelles IBC Act 2016 has AML requirements; the Seychelles AML Act 2020 has its own AML requirements. Both target the same compliance domain. Lex composes the resulting fibers using `verdict_meet`. This is a pessimistic composition inside the Lex verdict layer: the most restrictive verdict wins. Compliance with one statute does not excuse non-compliance with another statute governing the same domain. On fibers whose domain is Applicable, this agrees with the neighbouring tensor algebra's Applicable-fragment meet. Mixed applicability is not collapsed into this Lex chain at the tensor layer; it is carried as provenance by the receiving tensor composition. A required Applicable domain passes only

when every required fiber in that domain lands at `Compliant`; a single `Pending` drops the whole domain to `Pending`, and a single `NonCompliant` drops it to `NonCompliant`.

A concrete example: evaluating an ADGM entity against the rules of the ADGM Financial Services and Markets Regulations 2015 produces a vector of `FiberResult` values, each with a verdict and a compliance domain. The composition groups these by domain and takes the meet within each domain. If the FSRA authorization rule returns `Compliant` but the capital adequacy rule returns `Pending`, the overall verdict for the entity's financial services compliance is `Pending`. If one exemption rule returns `Exempt` and another rule in the same domain is `NotApplicable`, the domain verdict is `NotApplicable`: the domain passes, but the record correctly states that the rule corpus does not uniformly attach to the entity.

**Authority-indexed verdicts.** For tribunal-relative evaluation, write `[T]v` for "tribunal `T` asserts verdict `v`," formally shorthand for the proposition `[T](VerdictIs(v))`. Given verdicts `[T_1]v_1` and `[T_2]v_2`, a cross-authority meet is defined only through a comparison tribunal `U` and bridge witnesses `w_1 : CanonBridge(T_1, U, VerdictIs(v_1))` and `w_2 : CanonBridge(T_2, U, VerdictIs(v_2))`. The meet is then realized as

$$[T_1]v_1 \wedge_U [T_2]v_2 := [U](\text{verdict\_meet}(v_1, v_2)).$$

Operationally, each verdict is first coerced into the common tribunal `U` using the supplied bridge witness; only then is the verdict meet computed. If no common bridge witnesses exist, the cross-authority meet is ill-typed and the disagreement remains explicit.

**Receipt algebra for admission hosts.** A downstream execution host should not have to inspect a rule evaluator's entire derivation tree merely to compose it with another evaluation. It must, however, receive enough structure that composition cannot hide an unresolved obligation or discretionary frontier. A Lex receipt is the finite object

$$R = (v, O, U, F)$$

where `v in V` is the verdict, `O` is the finite set of declared obligations, `U subseteq O` is the finite set not yet discharged, and `F` is the finite set of discretion holes still pending authority. Define receipt composition by

$$R_1 \otimes R_2 = (v_1 \wedge v_2,\ O_1 \cup O_2,\ U_1 \cup U_2,\ F_1 \cup F_2).$$

A receipt is `accepted` when `U = \emptyset` and `F = \emptyset`; it is `compliant` when it is accepted and `v = Compliant`.

**Theorem (receipt locality / no-laundering).** For all receipts `R_1, R_2`,

$$accepted(R_1 \otimes R_2) \iff accepted(R_1) \wedge accepted(R_2)$$

and

$$compliant(R_1 \otimes R_2) \iff compliant(R_1) \wedge compliant(R_2).$$

Moreover, the verdict of `R_1 \otimes R_2` is the greatest lower bound of `v_1` and `v_2` in the verdict chain.

*Proof.* The verdict component is the meet in the Heyting chain, so the greatest-lower-bound property is the meet universal property. The obligation and frontier components compose by finite union; a union is empty iff each operand is empty. Since `Compliant` is the top element and meet is minimum in the legal-restrictiveness order, `v_1 \wedge v_2 = Compliant` iff `v_1 = Compliant` and `v_2 = Compliant`. The Rocq mechanization is `lex/formal/coq/Lex/ReceiptAlgebra.v`; `accepted_compose_iff`, `compliant_compose_iff`, and `admission_locality` close with `Qed`, and `Print Assumptions admission_locality` reports closed under the global context.

**Strict-priority disclosure.** The proposition below assumes strict priorities $p_1 > \cdots > p_n > p_0$. The current admissible fragment total-orders exceptions by `(priority, source-position)`, so equal numeric priorities do not trigger a meet in the mechanized evaluator; the earlier source position wins after sorting. A future equal-priority meet policy is a target extension and would require a separate proof that tied firing bodies compose by verdict meet without changing the evaluator agreement theorem.

**Proposition (priority evaluator agreement).** Let a defeasible rule have base body $b_0$ and exceptions $(g_1, b_1, p_1), \ldots, (g_n, b_n, p_n)$ ordered by descending priority $p_1 > \cdots > p_n > p_0$. For a context $c$, define

$$\nu_i(c) = \begin{cases} b_i(c) & \text{if } g_i(c) = \texttt{True} \text{ and } g_j(c) = \texttt{False} \text{ for all } j < i, \\ \top & \text{otherwise,} \end{cases}$$

and

$$\nu_0(c) = \begin{cases} b_0(c) & \text{if } g_i(c) = \texttt{False} \text{ for all } i, \\ \top & \text{otherwise.} \end{cases}$$

Then

$$\text{defeat}(r)(c) = \nu_0(c) \wedge \nu_1(c) \wedge \cdots \wedge \nu_n(c).$$

*Proof sketch.* Exactly one normalized clause contributes a value below $\top$. If some exception guard is true, let $k$ be the least index with $g_k(c) = \texttt{True}$; then $\nu_k(c) = b_k(c)$ and every other $\nu_i(c) = \top$. If no exception guard is true, then $\nu_0(c) = b_0(c)$ and every exception clause is $\top$. Since $\top = \texttt{Compliant}$ is the identity for meet, the meet of the normalized family returns exactly the verdict selected by the priority evaluator.

**Open obligation (pack-rewrite preservation).** Let $P \to P'$ be a pack evolution witness. If a `Time_1` verdict derivation under $P$ produces $\delta : P \vdash e \Downarrow (t, v)$, then re-evaluation under $P'$ produces a fresh derivation $\delta' : P' \vdash e \Downarrow (t', v')$ such that the semantic history is additive:

$$\text{History}' = \text{History} \cup \{(P, t, v, \delta), (P', t', v', \delta')\}.$$

The old verdict is preserved as the frozen historical derivation under $P$; the new verdict is a distinct `Time_1` derivation under $P'$. No rule overwrites $(P, t, v, \delta)$ with $(P', t', v', \delta')$.

*Proof target.* Re-evaluation should consume the same `Time_0` facts together with a new pack witness, producing a new derived judgment rather than mutating the old one. The missing proof must show that temporal stratification blocks any retraction from `Time_1` back into frozen history and that the semantics of pack evolution is append-only.

### 7.5 The fill() Round-Trip: A Discretion Hole Reaches an Authority

This example traces the complete lifecycle of a typed discretion hole, from first suspension to authorized resolution. Consider a compliance evaluator checking whether a newly appointed director of an ADGM-registered entity satisfies the "fit and proper" requirement under FSMR 2015 s.38.

**Step 1: Evaluation reaches the hole.** The evaluator runs the cross-jurisdictional rule from Section 7.2. For the ADGM branch, it resolves the `fit_and_proper_status` accessor against the entity's data. The director's status is `FitAndProperPending`, neither `FitAndProperSatisfied` nor `FitAndProperUnderReview`. The match falls through to the wildcard branch, which contains:

```
?fp_adgm : ComplianceVerdict
  @ authority ADGM.FSRA
  scope { jurisdiction: ADGM, entity_class: AuthorizedFirm }
```

The evaluator cannot reduce further. The term is well-typed and produces the `discretion(ADGM.FSRA)` effect. Evaluation suspends at this point with a structured **fill request**.

**Step 2: The runtime generates a fill request.** The evaluation runtime emits a fill request specifying exactly what is needed:

```
FillRequest {
  hole_id:    "fp_adgm",
  type:       ComplianceVerdict,      , one of NonCompliant, Pending, NotApplicable, Exempt,
  authority:  ADGM.FSRA,             , only FSRA personnel can fill this
  scope:      { jurisdiction: ADGM, entity_class: AuthorizedFirm },
  context:    {                       , the facts evaluated before suspension
    entity_id:    "entity-7f3a",
    director:     "John Smith",
```

```
    status_field: FitAndProperPending,
    rule_ref:     "FSMR 2015, s.38"
  },
  partial_derivation: <hash of the derivation trace up to the hole>
}
```

This request is routed to the responsible authority as a pending step in the evaluation. The suspended derivation records the exact answer type, responsible authority, and legal scope. No approximation enters the term.

**Step 3: A human fills the hole with a PCAuth witness.** An FSRA-authorized compliance officer reviews the director's qualifications and supplies a judgment. The filling carries a value-indexed witness whose signature commits to the specific verdict (`Compliant`):

```
fill(fp_adgm, Compliant, {
  quorum:     1,
  signers:    [did:example:officer-9b2c],
  depth:      0,
  authority:  [AuthorityChain(ADGM.FSRA, did:example:officer-9b2c, 0)],
  scope_ok:   ScopeWitness { jurisdiction: ADGM, entity_class: AuthorizedFirm, value: Complia
  timestamp:  2025-11-15T09:30:00Z,    , Time_0: frozen historical fact
  request_hash: Hash(FillRequest("fp_adgm", Compliant, pack_adgm_fsmr_2015, ctx_kyc_2025_11))
  pack_digest:  pack_adgm_fsmr_2015,
  context_digest: ctx_kyc_2025_11,
  anchor:     LinkedTimestamp(Hash(PCAuthPayload(...)), 2025-11-15T09:30:00Z),
  signatures: [
    Ed25519Sig(
      did:example:officer-9b2c,                                          , signer
      ADGM.FSRA,                                                          , authority
      "fp_adgm",                                                         , hole id
      Compliant,                                                         , filled verdict
      Hash(FillRequest("fp_adgm", Compliant, pack_adgm_fsmr_2015, ctx_kyc_2025_11)),
      Applied(PrecedentRef("adgm/fsra/fit-and-proper/2024-09")),         , mode
      Some("Applied FSRA fit-and-proper panel guidance from 2024-09"),   , justification
      LedgerRef("ledger://adgm/fsra/fp/2025-11-15/9b2c"),                , ledger ref
      pack_adgm_fsmr_2015,
      ctx_kyc_2025_11)
  ]
})
```

**Step 4: The derivation completes.** With the hole filled, the `discretion(ADGM.FSRA)` effect is discharged. The term reduces to `Compliant`. The full derivation trace records: "Steps 1-4 were mechanical evaluation of FSMR 2015 rules. Step 5 was a discretionary judgment by FSRA compliance officer `did:example:officer-9b2c` at 2025-11-15T09:30:00Z, supplying `Compliant` for the fit-and-proper determination, tagged `Applied` with a public justification ledger entry. PCAuth signature: `0x7a3f...`."

Every downstream consumer, the entity's compliance passport, a regulatory audit, a cross-jurisdictional corridor evaluation, can inspect this trace and distinguish the mechanical from the discretionary. The evaluator remains inside its typed authority boundary. The human judgment is typed, scoped, and authenticated under the stated key, credential, and bulletin assumptions.

### 7.6 Case study: ADGM↔DIFC commit

This case study fixes a single corridor event: an ADGM-licensed asset manager commits a sanctions-screened transaction with a DIFC counterparty. The companion Op paper develops the compiled bytecode, the lock witness, and the session-typed commit protocol for the same scenario under the same section title. The present concern is the Lex layer: which obligations are mechanical, which remain jurisdiction-indexed, and exactly where human judgment enters.

The two jurisdictions do not share a single sanctions predicate. The ADGM side screens against OFAC plus the ADGM list; the DIFC side screens against the UN consolidated list plus the DIFC list. Each rule is local and fail-closed:

```
let adgm_sanctions =
  lambda (tx : CorridorTx ADGM) [sanctions_query].
    match (screen OFAC tx.counterparty,
           screen ADGM_List tx.counterparty)
      return ComplianceVerdict with
    | (Clear, Clear) => Compliant
    | _ => NonCompliant

let difc_sanctions =
  lambda (tx : CorridorTx DIFC) [sanctions_query].
    match (screen UN_List tx.counterparty,
           screen DIFC_List tx.counterparty)
      return ComplianceVerdict with
    | (Clear, Clear) => Compliant
    | _ => NonCompliant
```

Sanctions mutual recognition is therefore not a coercion rule. Each side must evaluate its own lists; the bridge can only transport agreement once both local checks have passed.

The KYC layer contains the non-mechanical determinations:

```
?fit_and_proper_adgm : ComplianceVerdict
  @ authority ADGM.FSRA
  scope { jurisdiction: ADGM, role: AssetManager }

?approved_person_difc : ComplianceVerdict
  @ authority DIFC.DFSA
  scope { jurisdiction: DIFC, role: Counterparty }
```

The first hole marks the ADGM "fit and proper" judgment; the second marks the DIFC "approved person" judgment. Neither hole can be discharged by the other jurisdiction. Each requires a local PCAuth witness naming the authority, scope, digest, and timestamp of the fill.

The reporting obligation is mechanical again. Once the commit event `tx.commit_time` is fixed as `Time_0`, the corridor pack derives a legal deadline in `Time_1`:

```
let report_deadline : Time_1 =
  derive_1(tx.commit_time,
    corridor_reporting_window { period: 1_business_day }) in
match (tx.adgm_report_filed_by report_deadline,
       tx.difc_report_filed_by report_deadline)
  return ComplianceVerdict with
| (True, True) => Compliant
| _ => NonCompliant
```

The deadline is not a mutable clock value. It is a derived legal consequence of a fixed historical event, so the temporal stratification is doing real work in this example rather than serving as notation.

Let `CommitReady(tx_id)` be the proposition that the transaction may be committed. ADGM and DIFC assert it under their own tribunals. The `MutualRecognition` bridge below is *illustrative, not a current ADGM-DIFC framework*: at the time of writing, ADGM-DIFC cross-regulator cooperation covers data-protection adequacy, IOSCO MMoU information sharing, and a cross-border fund-licensing pathway; it does not transport fit-and-proper or commit-readiness determinations. The example uses a hypothetical bridge to show the typing discipline that a real cross-recognition instrument would have to supply:

```
adgm_ready : [ADGM] CommitReady(tx_id)
difc_ready : [DIFC] CommitReady(tx_id)
MutualRecognition :
  CanonBridge(ADGM, DIFC, CommitReady(tx_id))    -- hypothetical; see note above
```

With the bridge witness, the corridor may coerce one local judgment into the other tribunal:

```
coerce[ADGM => DIFC](adgm_ready, MutualRecognition)
```

The witness does not erase disagreement. If either sanctions rule fails, either discretion hole remains unfilled, or the bridge witness is absent, the coercion is ill-typed and the commit is blocked.

The proof obligations divide cleanly. The public executable checker covers the mechanical sanctions fragment; the filled-hole compilation case consumed by the companion Op paper is an Op-side admissible skeleton over already verified PCAuth entries, not a claim that the shipped Lex admissibility checker accepts raw `Hole` or `HoleFill` terms. The full-calculus trace records the hole lifecycle, and the admission envelope is the proof-carrying boundary that links the filled trace to the compiled payload. SMT discharges the finite arithmetic and equality side conditions for the reporting deadline and corridor identifiers. PCAuth fills the two KYC holes and the `MutualRecognition` bridge witness. Under the operational assumptions stated in the companion Op case study, partial synchrony, EUF-CMA security of Ed25519, and correctness of the sanctions and deadline oracles, an accepted commit is sound and accountable: sound because every mechanical obligation reduces against the relevant pack, accountable because every discretionary step is signed, scoped, and replayable.

### 7.7 Multi-Feature Example: Defeasible, Temporal, and Discretionary

This example demonstrates three of Lex's four distinctive features in a single rule. Consider the UK Companies Act 2006 s.853A requirement for filing a confirmation statement (introduced by the Small Business, Enterprise and Employment Act 2015 in substitution for the former annual-return regime at s.854):

```
-- Rule: Confirmation statement deadline (CA 2006 s.853A, as introduced 2015)
defeasible
  lambda (ctx : IncorporationContext).
    let incorporation_date : Time_0 = ctx.incorporation_date in
    let pack_ca2006 : PackType =
      Pack(uk_ca2006, v2006, rules_ca2006, 2006-11-08) in
    let pack_sbee2015 : PackType =
      Pack(uk_ca2006, v2015, rules_sbee2015, 2015-05-26) in
    let s854_effective : Time_0 =
      EffectiveDate(rule_ca2006_s854, 2006-11-08) in
    let sbee2015_effective : Time_0 =
      EffectiveDate(rule_sbee2015_s854, 2015-05-26) in
    let original_deadline : Time_1 = derive_1(
      incorporation_date,
      rewrite_ca2006_s854 {
        pack: pack_ca2006,
        effective: s854_effective,
        period: 14_days,
```

```
        from: anniversary
      }
    ) in
    let amended_deadline : Time_1 = reevaluate(
      original_deadline,
      Rewrite(pack_ca2006 -> pack_sbee2015)
    ) in
    match ctx.review_period_end_after sbee2015_effective
      return ComplianceVerdict with
    | True => match ctx.confirmation_filed_by amended_deadline
        return ComplianceVerdict with
      | True => Compliant
      | False => NonCompliant
    | False => match ctx.confirmation_filed_by original_deadline
        return ComplianceVerdict with
      | True => Compliant
      | False => NonCompliant
  priority 0
  unless
    (guard:    lambda (ctx : IncorporationContext).
                 match ctx.late_filing_application return Bool with
                 | True => True
                 | _ => False,
     body:     lambda (ctx : IncorporationContext).
                 ?good_cause : ComplianceVerdict
                   @ authority UK.CcompaniesHouse
                   scope { jurisdiction: UK },
     priority: 1)
end
```

This rule combines three features:

1. **Defeasibility**: The base rule (priority 0) requires filing by the deadline. The exception (priority 1) allows late filing for "good cause." These are independent legal principles with explicit priority, *lex specialis* in action.
2. **Temporal stratification**: The `incorporation_date` is `Time_0`, a frozen historical fact. `s854_effective` and `sbee2015_effective` are also `Time_0` values: typed effective dates for the two rule packs. `pack_ca2006` and `pack_sbee2015` are first-class `PackType` values. `original_deadline` is the `Time_1` deadline derived under the 2006 pack, and

`amended_deadline` is the `Time_1` consequence obtained by `reevaluate` along the typed witness `Rewrite(pack_ca2006 → pack_sbee2015)`. The 2015 amendment did not change when anyone was incorporated; it changed which pack governs fresh evaluation after its own effective date.

3. **Typed discretion hole**: The "honest and reasonable conduct" determination (`?good_cause`) is a discretion hole. A verifier can determine mechanically whether the filing was on time (the base rule). If the entity is applying for relief in respect of the confirmation-statement filing, evaluation reaches the hole and suspends: the court's relief discretion under CA 2006 s.1157, which permits relief for directors' honest and reasonable conduct, is genuinely open-textured, and no mechanical predicate fixes the verdict. The hole specifies that the responsible UK authority must supply a `ComplianceVerdict`. The evaluator emits a fill request; the authority supplies the judgment with a PCAuth witness; the derivation completes.

No existing compliance language can express this rule with all three properties typed and tracked. Catala handles the defeasibility; the temporal stratification and discretion hole remain outside its surface. L4 handles neither. A conventional implementation in Python would nest the exception in an if-else, use a mutable datetime for the deadline, and hardcode a boolean for "good cause", collapsing all three distinctions into opaque control flow.

### 7.8 Temporal Stratification: Statute of Limitations with Tolling

This example shows temporal stratification in Lex syntax. Under Pakistan's Limitation Act 1908 s.14, the period spent in bona fide prosecution of a related proceeding is excluded from the limitation period:

```
-- Statute of limitations with tolling (Limitation Act 1908, s.14)
lambda (ctx : IncorporationContext).
  let cause_of_action : Time_0 = ctx.cause_of_action_date in
  let limitation_pack : PackType =
    Pack(pk_limitation_1908, v1908, rules_limitation_1908, 1908-08-30) in
  let s14_effective : Time_0 =
    EffectiveDate(rule_limitation_s14, 1908-08-30) in

  let base_deadline : Time_1 = derive_1(
    cause_of_action,
    rewrite_limitation_act_1908 {
      pack: limitation_pack,
      period: 3_years
    }
  ) in
```

```
let first_tolled_deadline : Time_1 = Toll(
  base_deadline,
  ctx.first_bona_fide_prosecution_start
) in

let stacked_tolled_deadline : Time_1 = Toll(
  first_tolled_deadline,
  ctx.second_bona_fide_prosecution_start
) in

match ctx.first_tolling_after s14_effective return ComplianceVerdict with
| False => match ctx.filed_by base_deadline return ComplianceVerdict with
  | True => Compliant
  | False => NonCompliant
| True => match ctx.second_tolling_applies return ComplianceVerdict with
  | True => match ctx.filed_by stacked_tolled_deadline
      return ComplianceVerdict with
    | True => Compliant
    | False => NonCompliant
  | False => match ctx.filed_by first_tolled_deadline
      return ComplianceVerdict with
    | True => Compliant
    | False => NonCompliant
```

The key point: `cause_of_action` is `Time_0`. It records when something happened in the world. `s14_effective` is also `Time_0`: a typed effective date for the tolling rule. `base_deadline`, `first_tolled_deadline`, and `stacked_tolled_deadline` are all `Time_1`. They record what the law says about the consequences of what happened. The first toll consumes the base deadline; the second toll consumes the already-tolled deadline. No step re-derives from `cause_of_action`. That is the whole point of making `Toll` a first-class typed constructor: `Toll(Toll(deadline_0, t_1), t_2)` remains in `Time_1`. The only permitted coercion from frozen history is still the `lift_0`-to-`Time_1` direction; there is no path from any tolled deadline back to `Time_0`, because a legal rewrite cannot un-happen a historical fact.

---

## 8. From Calculus to Artifact

A working implementation of Lex must provide, at minimum: a parser for the surface syntax; an elaboration pass producing an elaboration judgment `G ⊢ s ⇝ c : A` that resolves the prelude, assigns de

Bruijn indices, and produces a machine-checked certificate witnessing that every refinement predicate attached to a surface binding survives on the corresponding core term and every surface effect annotation is preserved in the core effect row; a bidirectional type checker for the admissible fragment; a runtime evaluator that reduces type-checked rules against entity data; decision procedures for finite-domain membership, threshold comparison, and boolean propositions; a proof-obligation extractor that walks the AST and emits structurally-derived obligations (exhaustive match coverage, defeasible-rule guard decidability, sanctions-check presence); and a pretty-printer for diagnostic output. A substring-based approximation of the elaboration certificate is unsound: the certificate must be produced by the elaborator as a side effect of the typed translation, not reconstructed from lexical markers on the two representations.

Two architectural observations bear on implementation. First, definitional equality in the admissible fragment reduces to weak head normal form reduction under finite fuel with a finite substitution bound; the type checker performs this reduction and structural comparison at each step. Second, the decision procedures are total: they return either a proved verdict with witness, a refuted verdict with counterexample, or an explicit "undecidable with reason" verdict. The compliance-evaluation pipeline never hangs on an undecidable sub-problem.

A tabular authoring surface, decision tables compiled to canonical Lex AST, captures the rule patterns practitioners most often write (a rule name, a jurisdiction, a legal basis, and a set of condition-verdict rows with priorities). The tabular surface compiles directly to the `defeasible lambda match` pattern without extending the core AST. It is an accommodation to authoring practice, not a logical extension.

An external SMT backend (in SMT-LIB2 format) can discharge proof obligations beyond the admissible fragment's decision procedures. When SMT is unavailable, the pipeline should fall through to `Unknown` rather than assuming a verdict.

The admissible fragment also compiles into the Op virtual machine's bytecode. The companion paper on Op states an adequacy theorem schema: the soundness direction is the present closed target of the stepwise preservation development, while completeness up to Op-specific operational terminals and full abstraction on compliance contexts remain open obligations. The companion paper presents the compilation, the proof sketches in the Plotkin 1977 LCF-adequacy tradition, and the Coq mechanization of the soundness direction.

---

## 9. Categorical Semantics

The preceding sections present Lex syntactically. This section states a semantic target for the dependent core by interpreting Lex in a Category with Families (CwF) in Dybjer's sense, equivalently a comprehension category with dependent products and sums in the sense of Seely and Jacobs. Contexts are objects, types are dependent families over contexts, and terms are sections of display maps. The target is conditional on

the bridge strictification and adequacy obligations below; it is not a claim that the full calculus semantics has been mechanized.

A categorical model of Lex consists of:

- A CwF `(Ctx, Ty, Tm, 1, .)` with comprehension objects `Γ.A`, display maps `p_A : Γ.A → Γ`, and generic terms.
- A comprehension-category structure in which reindexing along each display map has left adjoint `Σ_A` and right adjoint `Π_A`, so dependent sums and products are interpreted as adjoints.
- Universe objects interpreting `Type_l`, closed under `Π` and `Σ`, with cumulativity.
- A proposition subfibration `Prop(Γ) ⊆ Ty_0(Γ)` whose fibers are Heyting algebras.
- For each tribunal `T`, an indexed adjunction `Tribunal-Assert_T ⊣ open_T`, natural in `Γ`. The tribunal modal is the indexed endofunctor `[T]_Γ = open_T ∘ Tribunal-Assert_T`.
- A temporal index category `[1] = (0 \to 1)`: the poset with two objects `0 < 1`, viewed as a thin category with a unique non-identity arrow `0 \to 1`. This is the walking arrow (equivalently, the nerve of the partial order `0 < 1` truncated at dimension 1); it is a *direct* category in Reedy's sense (Reedy 1974; cf. Hofmann 1995 on direct type dependency), with a strictly increasing degree function. `Time_0` is interpreted in the fiber over `0`, `Time_1` in the fiber over `1`, `lift_0` in the action of the arrow `0 \to 1`, and the asymmetry, no arrow `1 \to 0`, encodes temporal non-regression at the level of the indexing category itself. Using a direct (asymmetric) indexing category, rather than a monoid-graded or group-graded one, is load-bearing: a group grading would supply an inverse and readmit the `Time_1 \to Time_0` move that the typing invariant forbids.

The judgment interpretation is:

- ⟦`Γ`⟧ is an object of the CwF.
- ⟦`Γ ⊢ A : Type_l`⟧ is an element of `Ty_l(⟦Γ⟧)`.
- ⟦`Γ ⊢ e : A`⟧ is an element of `Tm(⟦Γ⟧, ⟦A⟧)`.
- ⟦`Γ, x : A`⟧ = `⟦Γ⟧.⟦A⟧`.
- ⟦`Pi(x : A). B`⟧ = `Π_⟦A⟧(⟦B⟧)` and ⟦`Sigma(x : A). B`⟧ = `Σ_⟦A⟧(⟦B⟧)`.
- ⟦`[T] A`⟧ = `open_T(Tribunal-Assert_T(⟦A⟧))`.
- ⟦`Time_i`⟧ is the value of the temporal presheaf at object `i` of the walking-arrow index category `[1] = (0 \to 1)`; reindexing along the arrow `0 \to 1` is the denotation of `lift_0`, and there is no reindexing functor in the opposite direction.

Substitution is reindexing in the CwF, so the operational equalities of the admissible fragment are interpreted as equalities of sections. The local bridge category from Section 4.6 is the fiberwise shadow of a global bicategory of authorities and bridges in Bénabou's sense: authorities are objects, bridge packages are 1-cells, and bridge refinements are 2-cells.

## 10. Denotational Model

We now specify a concrete model target inside a presheaf topos. For each fixed interface `A`, let `AT_A` be the category of authority-contexts, under the smallness hypothesis that the deployed authority labels, bridge witnesses, and temporal strata are drawn from a fixed small universe and under the bridge strictification/quotient obligation stated in §4.6. This hypothesis is structural: `AT_A^op → Set` is a presheaf topos only when `AT_A` is a small category. A multi-interface context is interpreted by a fibration of these authority-context categories over interfaces; the single-category notation below suppresses that index only inside one fixed interface.

- Objects are pairs `(auth, t)` where `auth` is a tribunal or authority label and `t ∈ {0,1}` is a temporal stratum.
- Morphisms are oriented opposite to object-language transport so that presheaf reindexing has the Lex direction: `Hom_AT_A((auth',t'),(auth,t)) := CanonBridge(auth,auth',A) x Hom_[1](t,t')`, where `Hom_[1](0,0)`, `Hom_[1](1,1)`, and `Hom_[1](0,1)` are singleton sets and `Hom_[1](1,0)` is empty. Thus a bridge `b : CanonBridge(T1,T2,A)` appears as a base morphism `(T2,t) → (T1,t)`, and the presheaf action along that morphism is a map `⟦[T1]A⟧ → ⟦[T2]A⟧`, matching `coerce[b]`.
- Composition is componentwise in the reversed base orientation: the bridge component composes as object-language bridges compose, while the presheaf arrow points in the opposite direction.

The identity and associativity laws for `AT_A` are inherited from the bridge bicategory only after the strictification/quotient obligation of §4.6 is discharged, or else carried by explicit bicategorical coherence. In the strictified case, the presheaf category `AT_A^op → Set` is a topos and therefore locally cartesian closed. It carries the canonical presheaf CwF: contexts are presheaves `E`, types over `E` are display maps `p : A → E`, and terms are sections `s : E → A`. Reindexing is pullback, while `Π` and `Σ` are the right and left adjoints to pullback along display maps. In particular, every closed Lex type at interface `A` is interpreted as a presheaf `Ob(AT_A) → Set`, and every open type is a dependent presheaf over the presheaf interpreting its context.

Base data types are constant presheaves. Compliance verdicts are interpreted in the constant presheaf `ΔH_verdict`, where `H_verdict = {NonCompliant < Pending < NotApplicable < Exempt < Compliant}` is the five-element Heyting chain of §7.4. Because Heyting structure in a presheaf topos is computed pointwise, `ΔH_verdict` carries internal Heyting-algebra structure: the structure maps `∧, ∨, ⇒ : (ΔH_verdict)² → ΔH_verdict` are presheaf morphisms by naturality, since they are constant on objects and hence commute automatically with every reindexing functor along a morphism of `AT`. Meet, join, and the relative pseudocomplement of §7.4 transport to the presheaf model pointwise.

Two bottom elements must not be confused. The order-theoretic bottom of the verdict chain is `NonCompliant`: an inhabited verdict, a legitimate failed outcome. The computational bottom `bottom_comp` of §4.12 is a distinct empty type in `Type_0` whose denotation is the empty presheaf, not the bottom element of `ΔH_verdict`. The Sanctions-Dominance rule produces an inhabitant of `bottom_comp` from a proof of sanctions non-compliance, which under the denotational model means the enclosing computation has no section over any authority-context; by contrast, producing `NonCompliant` means the enclosing computation has a section that selects the bottom element of the verdict chain. Conflating the two would trivialise the sanctions hard block, because every `NonCompliant` verdict would become an absurd term.

The Lex verdict layer can be viewed as the product presheaf `(ΔH_verdict)^23`, with meet and implication componentwise inside Lex's own verdict chain. That object is not the full tensor-value algebra used by the neighbouring composition layer, because the tensor separates Applicable compliance grades from the `NotApplicable` and `Exempt` applicability markers. The interpretation extends by postcomposition with mechanised alignment lemmas to the neighbouring compositional layers: `lex_to_tensor : ΔH_verdict → V_tensor` is Qed-injective (`Lex/TensorAlignment.v`) and commutes with meet on the Applicable fragment, while the mixed-axis counterwitnesses record where provenance would be lost. `verdict_to_cs` and `cs_to_verdict` (`Lex/PropagationAlignment.v`) are Qed mutual inverses between `ΔH_verdict` and the propagation-graph `ComplianceState`; both directions preserve rank, meet, and order. The denotational semantics of the Lex verdict is therefore compatible with the Applicable-fragment tensor algebra and with the propagation-graph layer by stated transport lemmas, not by an isomorphism of the full mixed-axis tensor.

The proposition fragment is interpreted by subobjects in the presheaf topos. For a fixed tribunal `T` and `A : Prop`, `⟦[T] A⟧` is obtained by applying `Tribunal-Assert_T` to the subobject interpreting `A` and then opening it back with `open_T`. When `open_T` is pullback-stable and idempotent, its action on subobjects is a Lawvere-Tierney topology. Lex does not require that global idempotence: mutual-recognition bridges may be one-way or scope-limited, so the semantics takes the weaker indexed endofunctor as primitive and recovers Lawvere-Tierney structure only on those tribunal fibers where the opening operator is idempotent. For general `A : Type_i`, the semantics must act on display maps in the slice category over the context and preserve the CwF structure; this display-map action is part of the open tribunal-modal semantics obligation rather than a consequence of subobject semantics alone.

**Open obligation (adequacy of the current admissible fragment).** If `Γ ⊢ e : A` is derivable in the current `L_adm` fragment, then for every object `c` of `AT` and every environment `γ ∈ ⟦Γ⟧(c)`, the denotation `⟦e⟧_c(γ)` is defined and belongs to `⟦A⟧_c(γ)`. Moreover, if `e ⟶ e'`, then `⟦e⟧ = ⟦e'⟧`; hence if `e ⇓ v`, then `⟦e⟧ = ⟦v⟧`. This obligation excludes constructs that `L_adm` currently rejects, including unfilled discretion holes and full modal forms.

*Proof target (per-constructor cases).* The intended proof is by induction on the typing derivation. The current `L_adm` target covers the standard CwF cases admitted by the fragment, finite constructor matches, defea-

sibility, and effect-row cases. The full-calculus target is a second obligation: it adds dependent pairs where admitted, tribunal modals, temporal stratification beyond pointwise proposition formers, and discretion holes. Those full-calculus cases are listed below as semantic obligations, not as cases already inside current `L_adm`.

*Standard CwF cases.* These follow Dybjer (1996, §3) and Jacobs (1999, ch. 10) verbatim:

- **Variable (`G, x:A ⊢ x : A`).** ⟦`x`⟧ is the generic term over ⟦`A`⟧, namely the second projection out of the comprehension ⟦`G`⟧.⟦`A`⟧.
- **Sort (`G ⊢ Type_l : Type_{l+1}`).** Universe objects in the presheaf topos are well-known (Hofmann-Streicher 1998); cumulativity is upward inclusion.
- **Dependent function (`G ⊢ Pi(x:A). B : Type_max`).** Right adjoint to display-map pullback: ⟦`Pi(x:A). B`⟧ := `Pi_`⟦`A`⟧(⟦`B`⟧).
- **Lambda (`G, x:A ⊢ b : B` => `G ⊢ lam x:A. b : Pi(x:A). B`).** Currying through the right adjunction.
- **Application (`G ⊢ f : Pi(x:A). B`, `G ⊢ a : A` => `G ⊢ f a : B[a/x]`).** Counit of the adjunction composed with substitution.
- **Let (`G ⊢ e : A`, `G, x:A ⊢ b : B` => `G ⊢ let x:A := e in b : B[e/x]`).** Reindexing by the section corresponding to `e`.

*Lex-specific cases.* Each requires its own argument under the presheaf interpretation:

- **Dependent match (Match-Dep).** For each constructor `C_i` of `e`'s type `T`, the branch body `b_i` denotes a section of the pulled-back motive `P[C_i(xs)/x]` over the appropriate sub-presheaf. The match denotation is the pointwise-defined coproduct injection: at each authority-context `c`, evaluate the scrutinee in fiber `c`, identify which constructor it matches by finite case analysis (Lex's admissible fragment restricts match to prelude constructor types with known finite variants; this finite-variant restriction is what makes the pointwise case-analysis well-defined in `Set`), and select the corresponding branch denotation. Pattern-arity shifts are handled by reindexing the branch context against the constructor-extended display map; this is the same mechanism that the typing rule's pattern-arity shift (§4.2 structural defect remark) addresses syntactically.
- **Defeasibility (Defeasible).** A defeasible term denotes a section of the verdict presheaf computed by priority-ordered evaluation: at each context `c`, evaluate every exception guard pointwise, identify the highest-priority firing exception (priority is a natural-number total order, decidable pointwise), and return that exception's body denotation if any fires, the base body denotation otherwise. The agreement with the priority evaluator (Proposition in §7.4) holds pointwise. This is *not* a standard CwF case: standard CwFs do not include priority-graph dispatch; the case requires showing that the pointwise-finite priority resolution is well-defined and natural in `c`, which it is because the priority graph is a closed term with no `c`-dependence.

- **Tribunal modals as bridge-indexed transport (Tribunal-Form, Tribunal-Assert, Tribunal-Coerce, Tribunal-Open).** `[T] A` denotes the indexed endofunctor `open_T o Tribunal-Assert_T` applied to ⟦A⟧ for proposition interfaces, and a corresponding action on display maps for general types. Adequacy here requires the bridge-action laws stated as open obligations in §4.6: `coerce[id_T^A]` denotes the identity transformation, `coerce[b_{23} * b_{12}]` denotes the composite transformation, and bridge 2-cell equalities lift to equality of coercion actions. A standard CwF does not carry tribunal modals; this case introduces bridge-indexed action on coercion morphisms and the corresponding indexed Hom set into Type. Naturality of `coerce[b]` must be proved separately from bridge 2-cell congruence: for every `u : c → d` in `AT`, the square `⟦[T2]A⟧(u) o coerce_{b,d} = coerce_{b,c} o ⟦[T1]A⟧(u)` is the presheaf-morphism law. Bridge 2-cell congruence compares parallel bridges; it is not itself this reindexing naturality square.
- **Priority-graph defeasibility (specifically the `eval_defeasible` agreement).** The evaluator's pointwise behaviour at each authority-context `c` is the same priority-ordered scan as in the operational semantics (§4.4, §7.4), so verdict denotations agree on closed terms. The case specifically requires showing that the priority graph is closed under reindexing along bridge maps (a defeasible rule's priority structure does not depend on which authority is evaluating it), which holds because priorities are typed at `Nat` (a constant presheaf) and the priority-comparison is decidable pointwise. Standard CwFs do not include this case; the priority-evaluator naturality has to be checked.
- **Temporal stratification (Lift, Derive, Toll, Reevaluate).** The temporal sort `Time_i` is interpreted via the walking-arrow index category `[1] = (0 \to 1)` of §9: grade `0` is the fiber over `0`, grade `1` is the fiber over `1`, and the unique arrow `0 \to 1` is the denotational image of `lift_0`. `derive_1(t, w)` is the same arrow action paired with a rewrite-witness selector; `Toll(d, t_{tol})` consumes a Time_1 section and an additional Time_0 section to produce a new Time_1 section (its lifted Time_0 input rides the arrow `0 \to 1` before the constructor fires); `reevaluate(d, w)` is the metalevel pack-rewrite map lifted to a presheaf morphism within the fiber over `1`. The temporal non-regression lemma (§3.2) is exactly the statement that `[1]` contains no arrow `1 \to 0`; because `[1]` is a direct category, its reindexing functors on presheaves never invert the grade. This case is not present in any standard CwF treatment because standard CwFs do not stratify their type universe by a direct indexing category; Lex's use of a strictly asymmetric index is what makes the typing-level invariant transport to the denotational level.
- **Typed discretion holes (DiscretionHole, MechanicalHole, UnsettledHole, Fill).** Filled holes denote total sections over the support where a PCAuth witness has been recorded; unfilled holes are not terms of the admissible total fragment. A richer full-calculus model can represent `DiscretionHole(auth, h, S)` as a partial section, or equivalently as a section into a lift/partial-map monad: at authority-contexts `c` where the hole is filled, the section is defined; at contexts where the hole is unfilled, it is absent. `Fill` is the section-extension operation. `UnsettledHole(dom)` is the empty support (not fillable by any current authority). The discretion-hole case is not standard

CwF; adequacy for a language that includes unfilled holes must be restated over the filled support or in a partial-map CwF, rather than as total-section adequacy over every context.

*Effect-row cases.* Effect rows are interpreted as monoidal grading on the presheaf semantics: each row `rho` selects a sub-presheaf carrying terms whose effect annotation is contained in `rho`. Effect-Weaken is the inclusion morphism. Branch-sensitive markings carry an additional unlock obligation tracked at the presheaf level. These cases are syntactic monoidal reindexing and do not require new categorical machinery beyond the direct-category indexing of §9.

Preservation of denotation under reduction (the second clause of the theorem) is a second induction on the reduction relation using the CwF beta/zeta/iota equalities for the standard cases and the per-case denotational equations spelled out above for the Lex-specific cases.

Mechanization of the full presheaf adequacy theorem remains open beyond the already mechanized flat fragment. The Lex-specific cases above are sketched at the level of "what denotational structure is needed and why standard CwFs do not supply it"; each sketch is one paper-length proof obligation in its own right, not a single argument that closes by induction. This adequacy theorem is therefore open in the mechanization status table at the end of §5.1; the present sketch is a road-map of the cases that a full closure would have to discharge, not a proof.

**Full abstraction.** Full abstraction for the full Lex calculus remains open. Set-valued presheaves quotient proof-relevant operational artifacts that Lex deliberately preserves in certificates: two terms with the same extensional verdict but different PCAuth witnesses, different bridge witnesses, or different derivation traces can have equal denotations while remaining observationally distinct to any observer allowed to inspect the emitted proof bundle or the outstanding effect frontier. A fully abstract model must be proof-relevant, for example an enriched presheaf model or game semantics of certificate traces. The gap is exactly the gap between extensional verdict equality and provenance-sensitive observational equality.

**Conjecture (cut elimination for the admissible fragment).** Assume the two open conjectures: subject reduction for admissible reduction (open; see Open Problems §13) and strong normalization for the full admissible fragment with dependent matches and modals (Conjecture SN-admissible, §5). Under both, every admissible derivation of `Γ ⊢ e : A` reduces to a cut-free derivation of the same judgment.

*Proof strategy modulo the two open conjectures.* In the admissible fragment without modals, cuts are precisely the redexes created by composing an introduction with its consuming context: beta for `Pi`, zeta for `let`, iota for finite `match`, and the finite dispatch redex for defeasible selection. If tribunal modals are admitted into the cut-elimination fragment, three additional contractions must be included: `tribunal-open`, `tribunal-coerce-id`, and `tribunal-coerce-comp`, each with its own subject-reduction case. Contracting such a redex preserves the end judgment if subject reduction holds. Strong normalization rules out an infinite sequence of cut contractions, so repeated contraction terminates at a derivation with no remaining cut redexes. That terminal derivation is cut-free by construction. The flat admissible fragment

already discharges the normalization premise constructively (`FlatAdmissibleSN.v`); extending the same argument to the full admissible fragment with dependent matches and modals reduces to the two remaining conjectures plus the modal contraction cases, not to a new proof-theoretic idea. Cut elimination is therefore stated as a conditional conjecture rather than as a free-standing theorem because both of its premises are themselves open.

---

## 11. Related Work

### 11.1 Catala

Catala (Merigoux, Chataing, and Protzenko, ICFP 2021) is the closest prior work. Catala compiles French tax law to executable OCaml, using a default calculus for defeasible reasoning. The key innovation is that Catala code is interspersed with the legal text it formalizes, maintaining traceability between statute and implementation.

Lex shares Catala's commitment to defeasibility as a first-class language primitive, not a library pattern. The two systems diverge on three points:

1. **Types.** Catala has a conventional type system (ML-family, no dependent types). Lex has dependent types with a universe hierarchy. This matters when the type of a compliance rule depends on the jurisdiction: an ADGM entity context has different fields than a Seychelles entity context, and the type system must track this.
2. **Discretion.** Catala does not have discretion holes. When the law requires judgment, the Catala programmer must supply a value. The boundary between mechanical derivation and human judgment is invisible in the compiled OCaml. Lex's typed discretion hole makes this boundary explicit and auditable.
3. **Authority.** Catala does not model multiple interpreting authorities. A Catala program encodes one interpretation of one statute. Lex's tribunal modals allow the same rule corpus to carry multiple authority-indexed interpretations that may diverge.
4. **Compilation target.** Catala compiles to OCaml/Python for execution. Lex produces proof terms: typed derivation traces that serve as compliance certificates. The output is evidence and verdict.

Catala's design philosophy avoids dependent types for accessibility to legal professionals. The choice is principled. Lex's complexity is justified where the compliance domain requires it: cross-jurisdictional rules whose result type depends on the jurisdiction, rules where human judgment must be typed and audited, regimes where multiple authorities interpret the same statute differently. For single-jurisdiction flat rules with no discretion boundary, Catala's simpler approach is equally appropriate, and Lex's admissible frag-

ment is expressively comparable. Lex earns its additional complexity at boundaries: between jurisdictions, between authorities, between machine and human.

The Catala line matters as production evidence and language design. The ICFP paper establishes the default-calculus core (Merigoux, Chataing, and Protzenko, 2021), and the subsequent CNAF deployment shows that statute-linked compilation can carry the French family-benefits corpus in production. Lex inherits that insistence on traceability and extends the target object: dependent types for jurisdiction-indexed contexts, tribunal modals for authority-indexed verdicts, and pack evolution, meaning explicit transitions between rule packs over time.

### 11.2 L4 (CCLAW, Singapore)

L4 is a domain-specific language line for legislation and contracts developed at the Centre for Computational Law at Singapore Management University (Lim et al., 2021). L4 uses deontic logic (obligation, permission, prohibition) to model legal rights and duties, with a visual representation for stakeholder communication.

L4 targets *contractual reasoning*, what parties are obligated and permitted to do under an agreement. Lex targets *regulatory compliance*, whether an institutional action is permitted under applicable law. The structural difference is that contracts are bilateral (between parties who agreed to terms) while regulations are unilateral (imposed by an authority on all entities in its jurisdiction). This leads to different design choices: L4 needs deontic modalities for modeling voluntary obligations; Lex needs defeasibility for modeling the exception structure of statutes and authority modals for modeling the multi-jurisdictional character of regulatory law.

The sharper comparison is feature-level. L4 includes a Defeasible module, Hohfeldian primitives, and deontic operators for normative positions (Lim et al., 2021). Lex places typed priority, tribunal modals for authority-relative verdicts, and PCAuth-backed discretion holes into the core calculus. The two systems choose different primitive legal structure.

### 11.3 Defeasible Logic Programming

Governatori, Prakken, and Sartor represent three decades of work on defeasible legal reasoning. Sartor's "Legal Reasoning: A Cognitive Approach to the Law" (2005) provides the theoretical foundation for defeasible reasoning in law. Prakken and Sartor (1997) formalize the interaction between lex specialis and lex posterior as a two-dimensional priority ordering, a direct theoretical antecedent to Lex's separation of defeasibility (lex specialis) from temporal stratification (lex posterior).

Governatori's defeasible logic frameworks (Governatori et al., 2000; Governatori, 2005) provide computational semantics for defeasible rules with superiority relations. The key difference from Lex is typing discipline: defeasible logic programs are untyped. A rule can defeat any other rule, even one with a different

semantic domain. Lex's typing rule for defeasible rules (Section 4.4) requires all exception bodies to inhabit the same type as the base body, catching a class of errors that untyped defeasibility permits.

The two-axis separation of specialis and posterior is due to Prakken and Sartor (1997). Lex's contribution on this axis is the typing discipline, the **Defeasible** rule requires type agreement, Boolean guards, and an explicit effect-row join (Proposition on defeasible effect join), together with the temporal stratification that handles posterior syntactically rather than as a second priority dimension.

PROLEG, Satoh et al.'s Prolog-based legal reasoner for Japanese civil-procedure statutes, belongs in the same lineage. Its significance is that it treats statutory reasoning as executable logic-program execution rather than mere document markup; its limitation, from Lex's perspective, is that the judgment boundary, authority of interpretation, and temporal provenance all remain outside the type discipline. Lex keeps the executable-rule ambition but adds typed discretion holes, tribunal modals, and temporal stratification as internal logical structure rather than external workflow.

The broader semantic family is non-monotonic reasoning: Reiter's default logic (1980), McCarthy's circumscription (1980), McDermott and Doyle's non-monotonic logic (1980), and Brewka's cumulative default logic (1991). Lex's defeasibility should be read as a typed instance within that family: adding information can defeat an earlier conclusion, but defeat is restricted by type agreement, explicit priorities, and authority-indexed derivations rather than by an untyped consequence relation alone.

The earliest published anchor for the Governatori line is Governatori, Antoniou, Maher, and Billington's "A Flexible Framework for Defeasible Logics" (AAAI 2000). That framework emphasizes modular proof theories and parameterized superiority relations; Lex inherits the concern for structured defeat, but fixes a single typed discipline because the dominant failure mode in compliance programming is cross-domain misalignment, not choosing among proof calculi.

Typed holes have precedent in dependent type theories (Idris, Agda) as interactive-proof goals: a hole stands for a term the user has not yet supplied, its expected type is displayed, and the type checker reports obligations against its context. Lex's discretion hole is structurally similar, a placeholder with a specified type, but differs on three points: (a) the hole is associated with an institutional authority, not a proof obligation; (b) it is discharged at runtime by a human with a PCAuth witness rather than by an elaborator; (c) it is integrated with an effect system that tracks outstanding obligations across the entire derivation. Lex's typed discretion hole contributes a specific legal-judgment boundary: a value-indexed, authority-scoped, effect-tracked hole discharged by a PCAuth witness.

McCarty's early legal-representation work, especially "A Language for Legal Discourse" (1989), already treats legal rules as objects requiring their own representational discipline rather than informal annotations over general-purpose code. Lex is in that lineage, but replaces representation-by-convention with a typed calculus: legal categories, authority boundaries, and admissible rule forms are checked by the type system itself.

### 11.4 LegalRuleML

LegalRuleML (OASIS, 2013) is an XML interchange format for legal rules. It supports defeasibility, temporal aspects, and jurisdictional metadata as markup elements. LegalRuleML is a serialization format for legal rules expressed in other formalisms, not an executable language.

Lex may use LegalRuleML as a serialization format. The relationship is analogous to that between MathML and a computer algebra system: the interchange format records structure; the language provides semantics. LegalRuleML's type system is schema-level XML (string-typed attributes), not a dependent type theory. The typing guarantees that Lex provides (a discretion hole has a specific type; defeasible exceptions agree in type with the base rule; temporal strata are never confused; a tribunal verdict cannot be silently reinterpreted as another tribunal's) cannot be expressed in LegalRuleML. The two systems address different layers: LegalRuleML addresses interchange, Lex addresses meaning.

### 11.5 Akoma Ntoso

Akoma Ntoso (OASIS LegalDocML, 2018) is an XML standard for machine-readable legal documents: statutes, regulations, case law, and parliamentary texts. Its contribution is structural markup (provisions, sections, amendments, and their relationships to source documents) for legislative drafting, legal research, and publication pipelines. Akoma Ntoso is a document standard, not a logic. It provides no evaluation semantics, no type discipline, and no mechanism for distinguishing mechanical derivation from human judgment. The relationship with Lex is natural: Akoma Ntoso marks up the text of a statute, Lex encodes the compliance rules derived from that text, and the two cross-reference (a Lex rule cites the Akoma Ntoso paragraph identifier of the statutory provision it encodes). Neither subsumes the other. The union delivers traceability from published statute to evaluated compliance verdict.

### 11.6 Open Texture in Computational Legal Reasoning

Bench-Capon and Sartor ("A Model of Legal Reasoning with Cases Incorporating Theories and Values," 2003) directly address Hart's open texture in computational terms. They propose a model where legal rules have associated "theories" (background justifications) that guide resolution in the penumbral region. The discretion boundary is acknowledged: certain cases require judicial creativity that goes beyond rule application.

Lex's typed discretion hole gives a formal interface for the legal-judgment boundary: a hole whose type, authority, and scope are checked by the type system, discharged by a human with a cryptographic authorization witness, and integrated with an effect system that tracks the obligation across a derivation. Bench-Capon and Sartor identify the phenomenon but do not provide this combined typed interface. Their model represents open texture as theory selection, choosing among competing justifications. Lex represents it as a typed hole: theory selection addresses *how* the judgment is made; the typed hole addresses *where* in the derivation it is needed and *what type* of judgment is required. The two perspectives compose rather than compete.

### 11.7 The Isabelle/HOL Formalization Tradition

The use of proof assistants for legal formalization has precedent in Libal and Steen's work on formalizing EU regulations in Isabelle/HOL (2019). Their approach encodes regulations as higher-order logic propositions and uses Isabelle's proof engine to verify properties of the encoding. Araszkiewicz and Zurek (2015) similarly use Isabelle for formalizing Polish civil code provisions.

Lex differs in that the logic itself is designed for the legal domain (with defeasibility, temporal strata, and discretion as primitives) rather than encoding legal rules in a general-purpose logic. The tradeoff is generality versus domain fitness: Isabelle/HOL can encode any legal rule that can be stated in higher-order logic, but the encoding burden falls on the user. Every defeasible rule requires manual construction of the priority machinery. Every temporal distinction requires manual stratification. Lex's primitives make the common patterns lightweight, at the cost of restricting the expressible rules to those with the structure described in Section 3.

### 11.8 Normative Programming Languages

Dinesh, Joshi, Lee, and Stuckey's work on normative programming (ICLP 2008) introduces a framework for modeling regulatory compliance as constraint satisfaction over normative states (obligations, permissions, prohibitions). Their system supports defeasibility through a priority ordering and temporal reasoning through event-based state transitions.

Lex's temporal stratification is also orthogonal to the classical LTL/CTL canon of Pnueli (1977), Manna and Pnueli (1992), and Wolper (1983). Those systems reason about properties of execution traces. `Time_0` and `Time_1` instead mark provenance strata of legally significant facts. Lex proves the anti-retroactivity invariant that derived legal consequences cannot coerce back into historical attestations.

The key distinction is that normative programming operates on a flat state space (propositions that are obligated, permitted, or prohibited at a given time), while Lex operates on a typed term language with dependent types, effect tracking, and proof terms. This matters for compositionality: in normative programming, combining two regulatory regimes requires reasoning about their joint state space. In Lex, combining two regulatory regimes is a type-theoretic operation, checking that the types align and that effect rows compose. The dependent types provide compositional guarantees that flat normative states cannot.

### 11.9 Smart Contract Verification

The formal verification of smart contracts (Hildenbrandt et al., 2018; Bhargavan et al., 2016) has produced substantial work on formalizing the semantics of on-chain legal agreements. These approaches verify that a contract implementation satisfies a specification, but the specification itself is typically written in a general-purpose logic (K framework, F*) rather than a domain-specific compliance language.

Lex targets rule fidelity: does this compliance rule faithfully encode this statute? The answer requires domain-specific primitives (defeasibility for exception structure, temporal strata for retroactivity, authority

modals for jurisdictional indexing) that general-purpose verification frameworks must encode manually. Conversely, Lex leaves the concurrency and reentrancy concerns that dominate smart contract verification to the execution layer; compliance rules are evaluated sequentially against a snapshot of entity state.

On-chain compliance work that deploys defeasible estate-planning rules on Ethereum is a closer comparator than pure bytecode verification, because it treats the legal rules themselves as the object of execution rather than the low-level machine code. Lex differs at the validity boundary: verdicts are authority-indexed, discretion is supplied by PCAuth-backed humans, and no blockchain commitment is constitutive of legal force. A chain may anchor publication or evidence, but Lex does not bind interpretation to a committed ledger state.

### 11.10 Hart's Penumbra, Dworkin's Critique, and Computational Legal Reasoning

Hart (1961) frames open texture as a defining feature of legal language: rules have a core of determinate application and a penumbra of genuinely contested cases where the rule itself cannot fix the answer. Dworkin (1977, 1986) contests Hart's positivism while preserving the observation that law includes judgment sites which no mechanical predicate can resolve. Bench-Capon and Sartor (2003) place open texture at the center of computational legal reasoning and argue that explicit argumentation frameworks are the appropriate formalism.

Lex agrees with Hart and with Bench-Capon/Sartor that the penumbra is unavoidable. Its representational contribution is narrower than a full penumbra model: a Lex discretion hole is a scoped, authority-bound, temporally-indexed refinement site at which a legally recognized principal supplies a value satisfying a named predicate. The construction captures the authorized-human-input aspect of the penumbra. It does not capture the richer jurisprudential structure: reasoning by analogy to prior cases, weighing competing principles, reshaping a rule through penumbral decision. Those moves belong to case-law accounts (Ashley, 1990; Ashley and Brüninghaus, 2009) and value-based argumentation (Bench-Capon, 2003), outside the current typed surface. Section 13 names this as a residual: Lex's Hart story is a bounded penumbral surface, not a full penumbral reasoning account.

### 11.11 Temporal Logic and Legal Time

Pnueli (1977) and Manna-Pnueli (1992) establish linear-time temporal logic (LTL); Emerson and Clarke's branching-time logic CTL (1982) and Alur, Henzinger, and Kupferman's CTL* / TCTL for real-time systems (1993) complete the canonical temporal-logic toolkit. Governatori, Rotolo, Sadiq, et al. (2011-2019) apply these logics to temporalized deontic reasoning over contracts and statutes; Pace, Schneider, and Tabone (2020) use LTL variants for monitor synthesis from regulatory text; Maggi, Di Ciccio, et al. (2011-2020) use Declare (LTL over finite traces) for procedural compliance checking.

Lex uses a weaker fragment than LTL/CTL/TCTL. Its temporal discipline indexes observations by admissibility time and by decision time, and its only temporal modality is the past-polarity / future-polarity distinction with a weakening lemma. Lex contains neither *until*, *next*, nor branching modalities; it contains

no real-time metric operators; it does not verify temporal formulas against rule executions. The design target is decision-time admissibility of evidence, not behavioral verification of workflow traces. A full LTL/CTL/TCTL positioning applies once Lex rules compose into long-running workflows; Op, the bytecode lane, is the natural host for metric temporal verification. Within Lex proper, temporal polarity carries admissibility, not behavioral logic. Section 13 names this as a scoped residual.

### 11.12 Information-Flow Control and Temporal Stratification

Denning (1976) originates lattice-based information-flow security. Sabelfeld and Myers (2003), Pottier and Simonet (2003), and Austin and Flanagan's faceted-values line (2012) supply language-based non-interference technology. The closest analogue in Lex is the temporal non-regression typing invariant (Lemma in §3.2): Time_1 (derived legal time) cannot coerce back to Time_0 (frozen historical time), which is structurally similar to a one-direction information-flow constraint where future-derived data cannot leak into a past-typed slot. The labelling is binary along the time axis rather than a rich lattice of security levels. A full non-interference *theorem* in the Sabelfeld-Myers sense, stating that two well-typed terms differing only in their Time_1 components produce identical Time_0 observations under every well-typed context, is not stated in this paper; the temporal non-regression lemma is the structural precursor that such a theorem would build on, and §13 names the full non-interference theorem as an open direction. Treating decision-time admissibility as a non-interference invariant rather than as a reviewer-side checklist is the application target; the formal statement remains future work.

### 11.13 Modal Type Theory and Typed Delegation

Typed delegation belongs equally to programming languages and to administrative law. Modal type theory (Pfenning and Davies, 2001; Nanevski, Pfenning, and Pientka, 2008) explains how authority can index judgments; capability-based security (Miller, 2006) and authorization logic (Abadi, Burrows, and Lampson, 1993; Garg and Pfenning, 2009) model delegated principals and their scope. Lex uses a simpler executable core than the full modal-logic apparatus and makes the delegation tree operational: a path exists or it does not, a scope contains another or it does not, and revocation either leaves the path active or it does not. The trade-off is deliberate. Modal type theory's expressive power is not needed for the compliance surface Lex targets; what is needed is decidable witness validation with contagious revocation, and that is what Lex provides.

---

## 12. Scope of the Contribution

The contribution of this paper is a logic for regulatory compliance rules in which defeasibility, temporal stratification, authority-relative interpretation, and typed discretion holes are primitive. It also includes a proof-theoretic account of authorized judgment, a conditional categorical semantic target for the full calculus, a temporal rule-graph non-regression theorem, a pack re-evaluation source-preservation theorem,

a PCAuth quorum extraction theorem, a receipt algebra for admission hosts, a fail-closed admission envelope, a finite-observation event-union support lemma for discretion-hole reductions, a canonical strict function-bridge target, and an administrative WHNF kernel. The full admissible-fragment checking theorem remains conditional on the remaining full-calculus metatheory. The per-statement enumeration of what is closed, conditional, and open appears in the Mechanization Status appendix at the end of §5.1.

The admissible fragment is the fragment on which the operational story stands; the full calculus is the fragment on which the design and semantic story stands. The gap between them is the research agenda of Section 13: full normalization and type soundness for the modal and dependent fragment, constructive replacement of the single classical step in the current mechanization, extending admissibility to include modals under a stratification condition, and treating the Sigma and recursion forms that the admissible fragment currently excludes. A complete account of witness revocation and pack-relative re-evaluation is in the same agenda.

Two further limits remain. First, the paper is about semantic and metatheoretic structure, not an empirical study of large rule corpora. Second, the comparison with Catala, L4, and related systems is argued at the level of constructs and examples rather than proved by a formal translation theorem.

These limits bound the contribution without diminishing it. The design is sharper because the objects now have semantics and partial mechanization, but the paper does not pretend every theorem one would want for the full language is already closed.

---

## 13. Open Problems

**Full metatheory of the modal and dependent fragment.** The administrative WHNF kernel is Qed-closed, but the full admissible-fragment checking theorem still depends on preservation, unconditional progress, confluence, full-calculus WHNF bounds, and strong normalization for terms with dependent matches, tribunal transport, temporal operators, and delegated hole fillings. Confluence is open at the diamond-lemma level: `par_diamond_spec` and `par_star_diamond_spec` are named Prop specifications, not Qed theorems. `Confluence.v` does Qed-close the transport lemma `confluence_from_par_star_diamond : par_star_diamond_spec → confluence_spec`, so the remaining confluence work is concentrated in the parallel-reduction diamond itself. The Qed-closed progress theorem is conditional on `confluence_property` and `match_exhaustiveness_property`; Rocq also closes `progress_from_match_coverage : confluence_property → match_coverage_property → progress_property`, reducing the match side to construction of a finite coverage certificate while leaving the confluence premise explicit. Preservation for the full admissible fragment is not Qed-closed. Conjecture SN-admissible still marks the strongest normalization claim for the full fragment.

The closure sequence is exact. First, replace the position-dependent `subst_args` fold in `step_match_ctor_fire` with a simultaneous constructor-argument substitution primitive and prove its well-scopedness, shift-commutation, and substitution-composition lemmas. Second, use that primitive to discharge `par_subst_spec` and `par_shift_spec`, then prove `par_diamond_spec` and `par_star_diamond_spec`; the already-closed bridge then yields `confluence_spec`. Third, make finite constructor coverage a typing or admissibility invariant that constructs `match_coverage_property`, which already implies `match_exhaustiveness_property`. Fourth, finish preservation by closing substitution-preserves-typing and the remaining beta, zeta, constructor-fire, and binder-congruence cases in `Preservation.v` (wildcard match preservation is Qed-closed as `preservation_case_step_match_wild`). Fifth, derive premise-free progress by applying the closed confluence theorem and a checker-constructed coverage certificate to `progress_from_match_coverage`. Sixth, extend `administrative_whnf_bounded_reduction` to a full admissible-calculus WHNF theorem under a stated syntactic fuel bound. Seventh, extend `flat_admissible_sn_ext` from the flat affine fragment to Pi-types, dependent matches, effect rows, and filled-hole evidence; modal and temporal forms enter admissibility only after their stratification obligations are integrated.

**Fully constructive mechanization.** The core scaffold uses one classical lemma for priority-graph acyclicity, and the wider Lex tree still uses functional extensionality in the propagation graph. The former decidable-equality axioms for the mutually recursive `Term` / `Branch` / `Exception` syntax have been replaced by Qed-closed mutual decision procedures. Replacing the acyclicity step with a constructive strongly connected components procedure, eliminating the propagation graph's functional-extensionality dependency, and extending the current Qed-closed ledger fraction (sixteen of twenty-seven non-conjectural entries, including explicitly marked support kernels; see the Mechanization Status appendix at the end of §5.1) to the remaining categorical and operational layers, is open.

**Revocation dissemination across corridors.** The local semantics are fixed in this paper: a typed revocation `Revoke(cred)` invalidates any verdict whose fill/admission time is at or after the revocation time, and taints historical verdicts whose supporting credential was revoked only after filling. What remains open is corridor-wide dissemination: in a multi-zone execution, how quickly must a revocation propagate for independently verifying zones to converge on the same `Tainted/Invalid` tag?

**Revocation and pack-relative re-evaluation.** Lex now records quorum, delegation, and timestamp structure inside PCAuth, but it still needs a complete semantic policy for what happens when a delegated authority is revoked, expires, or is later overruled after a hole has been filled. The open question is how those revocations interact with repeal, tolling, and rule-pack migration without corrupting the frozen historical record.

**Priority against repeal.** Priority graphs and temporal operators are now both first-class. What remains open is a jurisdiction-indexed doctrine for cases in which a later repeal or superseding enactment conflicts

with an earlier but more specific rule. The language can represent both axes; the legal resolution policy is not uniform across jurisdictions and is not fixed here.

**Pack-indexed admissibility.** Pack objects, rewrite witnesses, and repeal terms are now typed in the core calculus. What remains open is integrating pack selection into the admissibility predicate. For the admissible fragment, the checker must show that only a finite pack history is relevant to any evaluation and that proofs of `Active(r, P, t)` do not introduce cycles through rewrite search.

**Pack-evolution soundness for the modal and temporal subsystem.** Section 3.3 defines typed packs, replay-carrying rewrite witnesses, and a Qed-closed re-evaluation step for derived-time closures. What remains open is the full proof that the same construction commutes with arbitrary temporal terms, temporal modals, tribunal modals, and the admissibility predicate once pack indices are internalized into `Time_{1,P}`. This is stated there as a conjecture.

**Semantics of nested discretion holes.** What happens when a rule contains a discretion hole whose type itself depends on the filling of another hole? The current formalization allows this syntactically but does not specify an evaluation order for hole fillings. A dependency-ordered filling protocol is needed.

**Compilation to ZK circuits.** A zero-knowledge proof that a compliance evaluation was performed correctly, without revealing entity data, would enable privacy-preserving regulatory reporting. The open obligation is to define a compiler from typed Lex evaluation traces to R1CS or PlonKish relations and prove circuit soundness preserving verdicts, types, obligation frontiers, and PCAuth bindings. This target is conditional on the source metatheory and WHNF-bound closure above; before those close, a circuit proves only the bounded trace relation it encodes. Groth16 and PLONK are natural candidate proof systems, but the paper does not claim a closed compiler.

**Full modals in the admissible fragment.** The current admissible fragment rejects all modal forms (temporal, tribunal). Extending admissibility to include modals while preserving decidability requires a stratification argument: modal operators must preserve acyclicity in the dependency structure. The temporal stratification check enforces the necessary invariant (Time_1 cannot coerce to Time_0) locally. Integration with the admissibility predicate remains open.

**Oracle termination as an authority claim.** When a rule invokes an external oracle, the obligation `terminates_by oracle O: depth ⩽ k` is discharged at the proof-checking boundary by a signed attestation from the oracle operator together with a runtime bound check: the evaluator aborts the oracle at depth `k` and records whether the oracle produced an answer before the bound. This is an authority-backed trust assumption (the operator signs a termination claim) plus runtime enforcement by the counter. A mechanized termination proof for a specific oracle class, for example ownership-chain traversal bounded by graph diameter, would strengthen the attestation from a signed claim to a checked derivation. This remains an extension target.

**Type inference.** The bidirectional discipline requires type annotations on lambdas (checking mode) and explicit annotations on terms that cannot be inferred. A unification-based inference algorithm for the admissible fragment would reduce annotation burden, but unification in the presence of defeasible rules and effect rows is nonstandard.

**Privacy-preserving proof transport.** The admissible fragment remains a plausible source for privacy-preserving regulatory reporting, but a sound compilation of Lex derivation traces to succinct proof systems remains open. The question is no longer whether the trace has enough structure; it is how to preserve the language's typing guarantees under circuit or proof-system compilation.

The remainder of this section names the programming-language research programs the paper does not discharge. Each subsection fixes the residual precisely, names the prior-art anchor the closure would build on, and identifies the scope of the research owed.

### 13.1 Bounded Legal First-Order Quantification

Jurisdictional rules routinely quantify over bounded domains: "every beneficial owner with a stake above 25 percent," "all parent entities in the ownership graph," "some authorized director in the resolution set." Section 4.2 introduces the bounded quantifiers `forall x : A in C. P(x)`, `exists x : A in C. P(x)`, and `exists! x : A in C. P(x)` ranging over `Collection(A)` with a finite enumeration, together with witness extraction `choose`; the Proposition on finite-collection admissibility and the Bounded Skolemization theorem at the end of §4.2 give the expansion into `and_fold`, `or_fold`, and finite uniqueness checks, along with the bounded-search discipline for `choose`. What remains open is twofold. First, decidability of quantifier elimination for common statute-derived predicate shapes: linear arithmetic over enumerated bounded domains admits Presburger procedures (Bradley and Manna, 2007), but the treatment has not been stated uniformly across jurisdictions nor integrated with the tribunal-modal and temporal strata of §§4.5-4.6. Second, a cross-cutting soundness result showing that temporal polarity and authority indexing thread through the bounded quantifiers without loss of information (the admissibility sketch argues this pointwise; a uniform theorem is owed). The higher-order meta-quantifier `Pi_meta` (§4.2) remains outside the admissible fragment; extending admissibility to it would require reflection over rule syntax and metadata rather than bounded iteration, and is a separate research target.

### 13.2 Curry-Howard Correspondence

Lex's type system classifies values, not proofs. A full Curry-Howard story (proofs-as-programs, propositions-as-types) is possible for a substantial sub-language: the recursion-free fragment over finite enumerations inhabits a decidable fragment of constructive type theory with proof-irrelevant propositions. A complete treatment would (1) isolate the propositional fragment (types of `Prop` sort; see §13.4) and show that it matches Martin-Löf type theory's propositional equality and dependent function types, (2) give an explicit Curry-Howard reading for witness validation, where a witness for hole `h` with fill `v` is literally a proof of the proposition `phi_h(v) ∧ authorized(h, signer)`

`∧ temporally-admissible(audit)`, and (3) show that refinement soundness is the constructive projection of this proof onto the refinement predicate.

### 13.3 Logical Relations and Parametricity

Standard parametricity (Reynolds, 1983; Wadler, 1989; Bernardy, Coquand, and Jansson, 2010) for Lex's executable core would give a relational abstraction theorem: any Lex-typed program is relationally parametric in the types not fixed by the prelude. Two corollaries follow: (1) temporal polarity as a parametric label, so that equivalence of two fills differing only in future-polarity content is preserved by every well-typed context; (2) authority scope as a relation, so that two signer identities with the same authority scope and interval are indistinguishable to every Lex-typed consumer. Both claims are defensible on syntactic grounds in the current core (they are adjacent to the non-interference and cascade-revocation theorems). Neither has been stated as an explicit abstraction theorem with a logical-relations proof.

### 13.4 Prop-Sort Discipline and Proof Irrelevance

A full formal treatment of `Prop` as a proof-irrelevant erased sort requires three commitments. Section 4.1 discharges each locally: elimination from `Prop` into computational sorts is restricted to singleton propositions (Or-Elim-Prop, Subset-Elim, and Squash-Elim each target `Prop`); propositional equality is the intensional Martin-Löf identity type with Id-Form, Id-Intro (refl), and J, used intensionally without univalence or higher-path structure; and the Theorem (Prop erasure) at the end of §4.1 states that `erase(-)` preserves typing from Lex into the erasure-subcalculus `Lex_erased` under the explicit empty-effect premise for erased Prop subderivations. What remains open is the global consistency proof for the combined rule set (Id + Prop-Irrelevance + Dec-LEM + Classic-LEM + Or-Elim-Prop + Subset-Form/Intro/Elim + Squash) and the stronger certificate-preserving erasure theorem for Prop evidence with non-empty effect rows. Section 4.1 already flags this as open: the combination is modelled on Coq's `Prop` plus `Classical_Prop.classic` (Werner, 1997) together with a stricter singleton-elimination discipline, but a standalone mechanized consistency proof for the exact rule set, including the interaction of `Subset` with `Squash` and the dependent eliminator `J` on `Prop`-sorted motives, is not discharged in this paper. The Rocq mechanization in `lex/formal/coq/Lex/PaperMechanization.v` has the erasure theorem as a paper-only sketch rather than Qed; closing it against the eager-shift context of §5.1 and the parallel-reduction scaffolding in `Confluence.v` is the obligation this subsection names.

### 13.5 Higher-Dimensional Equality and HoTT/Cubical Treatment

Lex has a one-dimensional propositional identity type: §4.1 introduces Id-Form, Id-Intro (refl), and the Martin-Löf eliminator J, used intensionally. No univalence axiom is assumed, no higher-path structure is introduced, and no proof-by-path-induction beyond J is admitted. What is absent is *higher-dimensional* equality: no path type equipped with interval structure (cubical `Path`), no univalence connecting equivalence and equality, and no 2-cells connecting parallel paths. Two motivations to revisit this. First, certificates as paths: a Lex certificate exchanged between rule evaluators is naturally a path in the space of well-

typed derivations, and a cubical or HoTT path-structure would supply principled reasoning for certificate composition and comparison beyond the intensional J already available. Second, revocation as a path-groupoid operation: cascade revocation updates a delegation forest, and the resulting witness space has a natural groupoid structure that cubical type theory could expose as iterated-path algebra rather than as an external relation. Neither motivation is urgent for the stated metatheorems; the present J-based identity suffices for the intensional reasoning the paper requires. The higher-dimensional extension is listed as a research program, not a filled gap.

### 13.6 Modal Soundness and Completeness for Tribunal Bridges

Lex carries a term-level tribunal modality `[T] A` (§4.6) with introduction (`Tribunal-Assert`), elimination (`Tribunal-Open`), and bridge-mediated coercion (`Tribunal-Coerce`); bridges act covariantly on modal evidence, and the canonical strict function-bridge action is Qed-closed. What remains open is the *modal logic* counterpart: a Kripke-style semantics with accessibility given by tribunal delegation chains, and soundness and completeness theorems against that semantics. The missing piece is the labelled Hilbert system per bridge class and the canonical-model argument that closes completeness against the term-level modal already in §4.6. The technical content is doable along the lines of Pfenning and Davies (2001) for judgmental S4 and Nanevski, Pfenning, and Pientka (2008) for contextual modal type theory; the adaptation to bridge-witnessed accessibility is a natural next paper.

### 13.7 Denotational Semantics and Full Abstraction

Lex carries an operational semantics (§§4-5) and a denotational semantic target. Section 9 gives the categorical reading in a category with families (Dybjer, 1996; Jacobs, 1999), and §10 specifies a presheaf-model target over a small strictified authority-context category `AT`, with adequacy obligations split between the current `L_adm` fragment and the full calculus. What remains open is twofold. First, mechanization of adequacy beyond the flat fragment: each Lex-specific full-calculus case is a paper-length proof obligation rather than a single induction, and the Mechanization Status appendix at the end of §5.1 records adequacy as open. Second, full abstraction is explicitly *not* claimed by §10: set-valued presheaves quotient proof-relevant operational artifacts (PCAuth witnesses, bridge witnesses, derivation traces) that Lex deliberately preserves in certificates. A fully abstract model would need to be proof-relevant, plausibly an enriched presheaf model or a game semantics over certificate traces, so that two terms agreeing on extensional verdict but disagreeing on provenance remain observationally distinct. That proof-relevant refinement is the open research target; it is the shape of the denotational story that would match the operational commitment the admissible fragment already makes.

### 13.8 Full-Fragment Metatheory Beyond the Admissible Core

The paper names progress and preservation as open obligations beyond the currently discharged fragments. Current Rocq evidence gives conditional progress and preservation scaffolding, not an unconditional full-fragment theorem. The admissible-fragment route is the finite, total route:

close `conv_eq_shift_compat_spec`, `par_diamond_spec`, `par_star_diamond_spec`, construct `match_coverage_property` for checker-accepted matches, full preservation, premise-free progress, full WHNF bounded reduction, and the logical-predicate normalization extension from `FlatAdmissibleSN.v`. A full Lex language that also accepts general recursion, higher-order witnesses, and unbounded audit chains needs a separate theorem family: fixed-point semantics with a well-foundedness side condition, plus progress-modulo-divergence separating legitimate suspension-without-fill from undefined behavior. The present paper does not collapse those two programs into one theorem.

### 13.9 Discretion-Hole Lifecycle Beyond Hart's Penumbra

Hart's penumbra is a static descriptive concept. Lex's typed discretion hole captures the static "who decides" question. The dynamic question, "how does the decision come to bind?", requires a richer lifecycle: (1) model the fill as a speech-act-with-audit (Austin, Searle) whose legal force depends on downstream acknowledgment, challenge, or affirmance, (2) represent appellate overrule as a typed retraction propagated to downstream derivations, and (3) support fill-level reasons (a Dworkinian principle structure, beyond raw predicate satisfaction). The present paper fixes the static typing surface; the richer lifecycle is a research program on its own.

### 13.10 LTL/CTL/TCTL for Rule Composition at Workflow Scale

As noted in §11.11, Lex's temporal discipline is an admissibility polarity, not a behavioral temporal logic. Composed Lex rules evaluated inside a long-running workflow (Op, the bytecode lane) should satisfy temporal properties beyond admissibility: "every hole eventually either fills or times out," "no revocation cascades twice for the same delegation" (safety), "every consent-withdrawal leads to a halt" (liveness-under-fairness). These are natural LTL/CTL/TCTL obligations on the composed system, not on the Lex language itself. The research program is to state the workflow-scale temporal obligations formally and connect them to Op's operational semantics; this belongs to a sibling paper on the Lex → Op boundary.

### 13.11 Conflict Resolution for Equal-Authority Hole Fills

The executable core admits at most one outstanding hole at a time: the semantics serializes fills by construction. A fuller conflict-resolution account is a research program on its own, because real legal workflows involve scenarios the executable core does not cover: (1) two delegated principals with overlapping authority scope producing contradictory fills for adjacent holes, (2) a tribunal-of-record issuing a fill later challenged by a different tribunal with overlapping subject-matter authority, and (3) a fill valid at its decision time retroactively voided by appellate overrule, cascading to downstream holes that consumed the fill. The research program is to define a partial order on authority certificates, state a conflict-detection judgment that fires when two witnesses at equal authority produce different fill values, specify a repair protocol that transitions to a `conflict(h, {w₁, w₂, …})` state dischargeable only by a higher tribunal dominating all conflicting witnesses, and prove that the repair protocol preserves refinement soundness and future-leak

non-interference under the extended semantics. The present paper commits only to the single-outstanding-hole semantics, under which the problem does not arise.

---

## 14. Conclusion

Compliance rules are programs, but they have lacked a language that records authority, defeasibility, legal time, and authorized human judgment as first-class typed structure. Lex now gives a firmer account of that structure: tribunal modals are interpreted over explicit bridge data; defeasibility is separated from control flow by priority graphs and a Heyting verdict lattice; typed discretion is carried by value-indexed PCAuth witnesses with quorum, delegation, revocation, and timestamps; and legal time is represented by `EffectiveDate`, `Repeal`, `Toll`, and pointwise admissibility-time temporal proposition formers rather than host-language mutation.

The calculus therefore has a sharper semantic story than a design proposal alone. It supports a partial Curry-Howard reading on the recursion-free constructive fragment over finite enumerations in which compliance derivations are proof terms (extension to the full calculus is open; §13.2), and it specifies a categorical semantic target in a category with families and a presheaf topos, conditional on raw bridge strictification, presheaf naturality, adequacy, and full-abstraction obligations. Its administrative WHNF kernel is Qed-closed, its temporal theorem proves that the temporal rule graph has no `Time_1`-to-`Time_0` path, its pack re-evaluation theorem proves that replay changes only the pack-relative witness and not the frozen source fact, its PCAuth quorum theorem proves that accepted discretion-hole witnesses expose enough distinct policy-authorized signer attestations over the exact payload, and its receipt/admission algebra gives hosts Qed-closed locality and fail-closed envelope theorems for composing observable obligations and frontiers; the Mechanization Status appendix enumerates what is Qed-closed, conditional, and open, while the worked case studies in Section 7 show that the language remains legible on concrete statutory rules and in metatheoretic summary.

What remains open is narrower than before but still substantive. The full modal and dependent fragment still needs complete normalization, confluence, and type-soundness proofs; the current mechanization should be made fully constructive; and the semantics of re-evaluation under witness revocation and pack evolution need to be closed. Those are research obligations around the edge of the language, not uncertainty about the basic object the language names.

The typed discretion hole remains the load-bearing boundary. It is the point where the language says, with exact type and exact authority, that the machine must stop. That boundary is not a patch over incompleteness. It is the formal statement of what part of administrative law is computation and what part remains judgment.

---

## References


Abadi, M., Burrows, M., and Lampson, B. (1993). "A Calculus for Access Control in Distributed Systems." *ACM Transactions on Programming Languages and Systems*, 15(4), 706-734.

Alur, R. and Henzinger, T.A. (1994). "A really temporal logic." *Journal of the ACM*, 41(1), 181-203.

Araszkiewicz, M. and Zurek, T. (2015). "Comprehensive Framework Embracing the Complexity of Statutory Interpretation." In *Legal Knowledge and Information Systems: JURIX 2015*, IOS Press.

Austin, T.H. and Flanagan, C. (2012). "Multiple Facets for Dynamic Information Flow." In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*. ACM.

Bellare, M. and Rogaway, P. (1996). "The Exact Security of Digital Signatures: How to Sign with RSA and Rabin." In *Advances in Cryptology: EUROCRYPT 1996*, Springer LNCS 1070.

Belnap, N.D. (1977). "A Useful Four-Valued Logic." In J.M. Dunn and G. Epstein (eds.), *Modern Uses of Multiple-Valued Logic*, 8-37. D. Reidel.

Bench-Capon, T.J.M. (2003). "Persuasion in Practical Argument Using Value-Based Argumentation Frameworks." *Journal of Logic and Computation*, 13(3), 429-448.

Bench-Capon, T.J.M. and Sartor, G. (2003). "A Model of Legal Reasoning with Cases Incorporating Theories and Values." *Artif. Intell. Law*, 11(2-3), 97-143.

Bernardy, J.-P., Coquand, T., and Jansson, P. (2010). "A Presheaf Model of Parametric Type Theory." In *Mathematical Foundations of Programming Semantics (MFPS 2010). Electronic Notes in Theoretical Computer Science*.

Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., and Yang, B.-Y. (2012). "High-Speed High-Security Signatures." *Journal of Cryptographic Engineering*, 2(2), 77-89.

Bénabou, J. (1967). "Introduction to Bicategories." In *Reports of the Midwest Category Seminar*, Springer LNM 47, pp. 1-77.

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

Birkhoff, G. (1948). *Lattice Theory*. American Mathematical Society.

Bradley, A.R. and Manna, Z. (2007). *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer.

Brewka, G. (1991). *Cumulative Default Logic*. Springer.

Cramer, R. and Damgård, I. (1996). "New Generation of Secure and Practical RSA-Based Signatures." In *Advances in Cryptology: CRYPTO 1996*, Springer LNCS 1109.

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

Dinesh, N., Joshi, A., Lee, I., and Sokolsky, O. (2008). "Reasoning about Conditions and Exceptions to Laws in Regulatory Conformance Checking." In *Deontic Logic in Computer Science (DEON 2008)*, Springer LNCS 5076.

Dworkin, R. (1977). *Taking Rights Seriously*. Harvard University Press.

Dworkin, R. (1986). *Law's Empire*. Harvard University Press.

Dybjer, P. (1996). "Internal Type Theory." In *Types for Proofs and Programs*, 120-134. Springer LNCS 1158.

Eilenberg, S. and Zilber, J.A. (1950). "Semi-Simplicial Complexes and Singular Homology." *Annals of Mathematics*, 51(3), 499-513.

Garg, D. and Pfenning, F. (2009). "A Logical Framework for Justified Belief." Technical Report CMU-CS-09-151, Carnegie Mellon University.

Grothendieck, A. (1971). "Catégories fibrées et descente." In *Revêtements Étales et Groupe Fondamental (SGA 1)*, Springer LNM 224, exposé VI.

Governatori, G. (2005). "Representing Business Contracts in RuleML." *International Journal of Cooperative Information Systems*, 14(2-3), 181-216.

Governatori, G., Antoniou, G., Maher, M.J., and Billington, D. (2000). "A Flexible Framework for Defeasible Logics." In *Proceedings of AAAI 2000*. AAAI Press.

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

Hart, H.L.A. (1961). *The Concept of Law*. Oxford University Press.

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.

Hofmann, M. (1995). "Extensional Concepts in Intensional Type Theory." PhD thesis, University of Edinburgh.

Hofmann, M. and Streicher, T. (1998). "The Groupoid Interpretation of Type Theory." In G. Sambin and J. Smith (eds.), *Twenty-Five Years of Constructive Type Theory*, 83-111. Oxford University Press.

Jacobs, B. (1999). *Categorical Logic and Type Theory*. Elsevier.

Johnstone, P.T. (1982). *Stone Spaces*. Cambridge University Press.

Johnstone, P.T. (2002). *Sketches of an Elephant: A Topos Theory Compendium*. Oxford University Press.

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

Levi, E.H. (1949). *An Introduction to Legal Reasoning*. University of Chicago Press.

Libal, T. and Steen, A. (2019). "NAI: Towards Transparent and Usable Semi-Automated Legal Analysis." In *Proceedings of the Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements (ARCADE 2019)*.

Manna, Z. and Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems*. Springer.

Martin-Löf, P. (1984). *Intuitionistic Type Theory*. Bibliopolis, Naples. Notes by G. Sambin from lectures given in Padua, June 1980.

McCarthy, J. (1980). "Circumscription: A Form of Non-Monotonic Reasoning." *Artif. Intell.*, 13(1-2), 27-39.

McCarty, L.T. (1989). "A Language for Legal Discourse." In *Proceedings of ICAIL 1989*. ACM.

McDermott, D. and Doyle, J. (1980). "Non-Monotonic Logic I." *Artif. Intell.*, 13(1-2), 41-72.

May, J.P. (1967). *Simplicial Objects in Algebraic Topology*. Van Nostrand, Princeton. Reprinted University of Chicago Press 1992.

Merigoux, D., Chataing, N., and Protzenko, J. (2021). "Catala: A Programming Language for the Law." *Proceedings of the ACM on Programming Languages*, 5(ICFP), Article 77, 77:1-77:29.

Miller, M.S. (2006). *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University.

Nanevski, A., Pfenning, F., and Pientka, B. (2008). "Contextual Modal Type Theory." *ACM Transactions on Computational Logic*, 9(3), Article 23, 23:1-23:49.

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.

OASIS. (2013). "LegalRuleML Core Specification Version 1.0." OASIS Standard.

Pfenning, F. (2001). "Intensionality, extensionality, and proof irrelevance in modal type theory." In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS 2001)*. IEEE.

Pfenning, F. and Davies, R. (2001). "A Judgmental Reconstruction of Modal Logic." *Mathematical Structures in Computer Science*, 11(4), 511-540.

Plotkin, G.D. (1977). "LCF Considered as a Programming Language." *Theoretical Computer Science*, 5(3), 223-255.

Pnueli, A. (1977). "The Temporal Logic of Programs." In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, 46-57. IEEE.

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

Prakken, H. and Sartor, G. (1997). "Argument-Based Extended Logic Programming with Defeasible Priorities." *Journal of Applied Non-Classical Logics*, 7(1-2), 25-75.

Reedy, C.L. (1974). "Homotopy theory of model categories." Unpublished manuscript, University of California, San Diego. Cited in Hovey, *Model Categories* (AMS, 1999), §5.2, as the source of the Reedy-category framework.

Reiter, R. (1980). "A Logic for Default Reasoning." *Artif. Intell.*, 13(1-2), 81-132.

Reynolds, J.C. (1983). "Types, Abstraction and Parametric Polymorphism." In *Information Processing 83*, IFIP, pp. 513-523. North-Holland.

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

Sangiorgi, D. (1998). "On the Bisimulation Proof Method." *Mathematical Structures in Computer Science*, 8(5), 447-479.

Sartor, G. (2005). *Legal Reasoning: A Cognitive Approach to the Law*. Springer.

Satoh, K., Asai, K., Kogawa, T., Kubota, M., Nakamura, M., Nishigai, Y., Shirakawa, K., and Takano, C. (2011). "PROLEG: An Implementation of the Presupposed Ultimate Fact Theory of Japanese Civil Code by PROLOG Technology." In *JSAI-isAI 2010*, Springer LNAI 6797.

Seely, R.A.G. (1984). "Locally cartesian closed categories and type theory." *Mathematical Proceedings of the Cambridge Philosophical Society*, 95(1), 33-48.

Skolem, T. (1920). "Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Beweisbarkeit mathematischer Sätze nebst einem Theoreme über dichte Mengen." *Videnskapsselskapets Skrifter, I. Matematisk-naturvidenskabelig Klasse*, 4, 1-36.

Wadler, P. (1989). "Theorems for Free!" In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA 1989)*. ACM.

Werner, B. (1997). "Sets in types, types in sets." In *Theoretical Aspects of Computer Software (TACS 1997)*, Springer LNCS 1281.

Wolper, P. (1983). "Temporal Logic Can Be More Expressive." *Information and Control*, 56(1-2), 72-99.

Wong, M.W., et al. (2022). "Overview of the CCLAW L4 project." In *Workshop on Programming Languages and the Law (ProLaLa 2022)*, co-located with POPL 2022.