The Calculus of Disposition

Processes, Types, and Witnessed Computation

Flyxion
Independent Researcher

September 2026

Complete draft

Contents

Preface

Every formal calculus of computation decides what a term is, what may happen to it, when two results count as the same, and what an observer is allowed to distinguish. These decisions are often introduced separately. They are not independent. A calculus that treats computation as rewriting will naturally ask which rewritten terms have the same normal form. A calculus that treats computation as communication will instead ask which interactions an environment can observe. A calculus that retains a history must ask a further question: which differences may be ignored without asserting that recorded events never occurred?

The calculus developed in this book begins from the last question. Its basic subject is not a value alone, nor a process considered apart from its past, but a commitment moving through a witnessed history. A commitment may be bound, transformed, verified, refused, or collapsed into an observation. Each such action changes what continuations remain admissible. None is permitted to revise the past that made it possible.

This orientation yields a system with three formal layers and one durable semantic object. A lambda layer computes values, predicates, transformations, reasons, and witnesses. A process layer organizes fresh commitments, interaction, choice, concurrency, and disposition. A type algebra separates reusable knowledge from linear rights to advance a commitment. An append-only ledger records the events from which the current disposition state can be reconstructed. The layers are kept distinct on purpose. Lambda terms compute values; processes consume and produce commitments, capabilities, evidence, and observations. A transformation has a functional type such as $f : A \multimap B$, while a process applies that transformation within the history of one particular commitment.

The principal theorem is therefore not merely that well-typed terms do not go wrong. It is that a commitment reaches a terminal disposition only by an authorized, causally complete, and durably witnessed history. We call this property *typed disposition safety*. The proof is cumulative:

parsing and α-invariance
$\Downarrow$
lifecycle, freshness, and history preservation
$\Downarrow$
authority, binding, and witness ancestry
$\Downarrow$
collapse soundness and terminal exclusivity
$\Downarrow$
replay and compensation
$\Downarrow$
typed disposition safety

The final proof is short because the book is arranged so that nearly all of its content has already been proved locally. In the present calculus most of the untyped spine turns out to rest on a single invariant, the *ledger path lemma* of Chapter 17: the records of each commitment, read in order, trace a legal path through its lifecycle ending at its current state. Binding integrity, witness ancestry, collapse soundness, terminal exclusivity, and refusal preservation are then corollaries rather than separate theorems. The type algebra of Part IV strengthens the spine in its final form: commitment uniqueness from linearity, lifecycle monotonicity from typestate preservation, and the impossibility of a premature collapse from the input type of collapse.

Two layers, one vocabulary

A reader who knows the canonical Spherepop kernel will notice that the disposition calculus uses its operator names with more specific meanings. In the kernel, Pop commits to one option among those available; Refuse records that an option is inadmissible without removing it from the history; Bind couples two elements without identifying them; and $\mathsf{Collapse}_c$ observes a history under a rule c. In the disposition calculus, pop introduces a fresh commitment; refuse is one of two terminal dispositions, available from every live state; bind associates a commitment with a value or capability; collapse produces an observation from a verified commitment under a declared rule k; and transform and verify appear as lifecycle stages.

The book's claim is that the disposition calculus is a discipline *over* the kernel rather than a separate calculus, and it is a claim to be proved, not assumed. Chapter 3 gives an explicit translation of disposition records into kernel histories and proves it in both directions at the level of ledgers: every disposition step becomes an admissible kernel step, and every kernel step of translated shape whose guard holds comes from a disposition step. Under that result the four kernel operators remain the causal basis; transformation, verification, and discharge are guarded compositions of them; the kernel itself enforces freshness and finality; and everything else the lifecycle requires lives in the guard. What the result does not establish is that the kernel performs the computations the guards check. That stronger, transparent reading is stated as a conjecture.

The two refusals agree on the property the whole corpus cares about: a refusal is recorded rather than erased. They differ in scope. Kernel refusal concerns an option and the history continues around it. Disposition refusal concerns a commitment's entire future, and a later reconsideration must create a distinct successor commitment rather than reopen the refused one.

Three comparative claims

Alongside the construction, the book tests three claims about the calculus's position, each of which could turn out false. Each is stated with what would settle it.

The first is that the calculus's nearest formal relative is the theory of event structures with conflict rather than the π-calculus or the lambda calculus. Every disposition world has a labelled event structure underneath it, together with annotations η carrying principals, reasons, witnesses, certificates, versions, and rules (Chapter 2). The claim is settled positively if the admissibility conditions can be expressed as structural constraints on such a structure over a finite alphabet, and negatively, with η as the identified residue, if some condition depends

irreducibly on evidence predicates.

The second concerns refusal. In its defensible form it says that recorded refusal is not represented by the standard failures model of CSP without adding an internal event or instrumentation. A CSP refusal set is a property of behaviour inferred from outside a process; a recorded refusal is authorized, reasoned, and witnessed data inside it. The stronger claim, that no counterpart exists in any form, would need a separation result, and until one is proved the book asserts only the weaker statement. It is settled by an encoding of recorded refusal into instrumented CSP together with a proof that uninstrumented failures equivalence identifies histories that differ only in their recorded refusals.

The third is that making collapse rule-indexed changes the theory of equivalence rather than relocating it. The observable label of a collapse is the pair $\langle k, o \rangle$, so equal payloads under different rules are distinct. The claim is settled by defining equivalence of processes relative to a rule, $P \approx_k Q$, and determining when it is a congruence, which classical equivalences arise as rules, and whether simultaneous observation of one history under several rules yields a result that van Glabbeek's linear-time/branching-time spectrum does not already state. If it yields none, the spectrum contains the substance, and Part VII says so.

One early piece of evidence bears on the first claim. Chapter 20 proves that replay is invariant under exchanging adjacent records of distinct commitments, so that a ledger determines its disposition state only up to a Mazurkiewicz commutation class. The causal order that the ancestry theorems need is therefore the dependence order of a trace, not the raw order of the ledger, and that is exactly the order an event structure makes primary.

What counts as a result

Positive comparative claims are stated relative to Gorla's encodability criteria and the standard of full abstraction; negative claims follow the model of Palamidessi's separation results. Spine theorems are stated with their hypotheses visible. The liveness result in particular holds only under fairness, responsive external dependencies, terminating transformations, and decidable verification, and is stated with all four. Three grades of assertion are kept apart: results proved here; standard results cited and used without re-derivation, marked where load-bearing; and conjectures or programmatic claims, collected in Appendix E.

The book also uses three levels of validation, and they are not interchangeable:

$$\text{exhaustive finite checks} \subset \text{untyped semantic proofs} \subset \text{typed universal guarantees.}$$

The laboratory of Chapter 2 enumerates every admissible ledger up to a bound and finds countermodels and specification errors; it proves nothing about larger worlds. The untyped proofs of Parts III, V, and VI establish the behaviour of every operational derivation. The typed metatheory of Part IV establishes that well-formed programs cannot even express several classes of duplicated, abandoned, or misindexed continuation. Each level catches things the next cannot state, and none substitutes for the one above it.

Status of this edition

Every chapter is drafted. The comparative results are summarized in Chapter 27, which answers the three claims above: the first positively with an identified residue, the second in its

weak form, and the third in a precise form that adds an orthogonal spectrum of equivalences on histories to the classical spectrum on processes. The open questions that remain are listed in Appendix E, chief among them the transparent reading of the kernel translation.

Part I

Orientation

Chapter 1

Computation with a Past

1.1 Four commitments of a calculus

A calculus is sometimes presented as syntax followed by reduction rules. That description omits two choices which are equally fundamental. Besides its terms and steps, a calculus fixes an equivalence and an observational boundary. The untyped lambda calculus, for example, begins with variables, abstractions, and applications; advances by beta reduction; compares terms through a selected convertibility or observational relation; and exposes a result through a normal form, a head form, or interaction with a context. Process calculi alter every component. Their terms denote persisting agents, their transitions carry labels, their equivalences compare behaviours, and their observers distinguish actions rather than returned values alone.

The coupling matters. If an observer sees only final values, two histories may be identified even when one passed through a refusal and the other did not. If refusal is itself observable, that identification becomes invalid. If an append-only record survives the computation, equivalence cannot mean literal equality of histories unless no abstraction is permitted. It must instead mean equality under a declared observation rule.

Definition 1.1 (Disposition)**.** *A* disposition *is an attributable and recordable determination of the admissible future of a commitment. A disposition may advance the commitment, refuse its continuation, or produce a terminal observation.*

This definition separates refusal from failure. A machine failure may prevent the production of any disposition. A refusal is already a result: it identifies a commitment, an authorized source, a reason or predicate, and a record which persists after the refusal has become historical.

1.2 State and history

The running configuration is written

$$C = \langle P, S, A, O, H \rangle.$$

Here P is a process, S is mutable computational state, A maps commitment identities to their current disposition states, O contains observations, and H is an append-only history. The components are kept separate because they answer different questions. The store says what is currently available to computation. The commitment map says which lifecycle rights are live.

The observation set says what has been made public under a collapse rule. The history says how those facts arose.

Two executions may therefore reach extensionally equal stores while remaining semantically different:

$$S_1 = S_2 \quad \text{but} \quad H_1 \neq H_2.$$

This inequality is not administrative residue. If H_1 contains a refusal and H_2 does not, a later observer may legitimately distinguish them.

Definition 1.2 (History extension)**.** *For histories H and H', write $H \sqsubseteq H'$ when H is a prefix of H'. A transition system is* historically monotone *when*

$$C \longrightarrow C' \quad \Longrightarrow \quad H \sqsubseteq H'.$$

Historical monotonicity does not prohibit correction. It prohibits correction by erasure. A later event may compensate for an earlier event, but the compensation is intelligible only because both remain available.

1.3 The lifecycle

A commitment has an identity c, content of type A, and a lifecycle state. At the untyped operational level we use

$$\begin{aligned} d ::= {} & \mathsf{open}(q) \mid \mathsf{bound}(q, b) \mid \mathsf{transformed}(b, v) \\ & \mid \mathsf{verified}(v, w) \mid \mathsf{verificationFailed}(v, \varphi) \\ & \mid \mathsf{refused}(r) \mid \mathsf{collapsed}(o). \end{aligned}$$

The live states are open, bound, transformed, verified, and verificationFailed. The terminal states are refused and collapsed.

Verification has two outcomes and two successor states. A positive result leads to verified, carrying a witness. A negative result leads to verificationFailed, carrying an authenticated certificate of the failure. The name is chosen with care: nothing has failed to execute. Verification has returned a certified negative result. The two branches are symmetric: successful verification does not collapse the commitment, and failed verification does not refuse it. Verification classifies evidence; an authority disposes of the continuation.

Refusal is permitted from every live state. This is essential. A reviewer may reject a binding, a transformation may be found inadmissible, or verification may have failed. If refusal were available only from open, these outcomes would become unrecorded dead ends rather than dispositions.

Let $\preceq_D$ be the least reflexive and transitive relation containing the constructive edges

$$\mathsf{open} \prec_D \mathsf{bound} \prec_D \mathsf{transformed} \prec_D \mathsf{verified} \prec_D \mathsf{collapsed},$$

$$\mathsf{transformed} \prec_D \mathsf{verificationFailed},$$

and a refusal edge from each live state to refused. From verificationFailed the refusal edge is the only one. The two terminal states are incomparable. Consequently, no ordinary lifecycle transition can turn a refusal into a collapse or a collapse into a refusal.

The lifecycle is small enough to tabulate. For each state s, let $\mathsf{Next}(s)$ be the set of actions that may follow it:

$$\begin{aligned} \mathsf{Next}(\mathsf{open}) &= \{\mathsf{bind}, \mathsf{refuse}\}, & \mathsf{Next}(\mathsf{bound}) &= \{\mathsf{transform}, \mathsf{refuse}\}, \\ \mathsf{Next}(\mathsf{transformed}) &= \{\mathsf{verify}, \mathsf{refuse}\}, & \mathsf{Next}(\mathsf{verified}) &= \{\mathsf{collapse}, \mathsf{refuse}\}, \\ \mathsf{Next}(\mathsf{verificationFailed}) &= \{\mathsf{refuse}\}, & \mathsf{Next}(\mathsf{refused}) &= \mathsf{Next}(\mathsf{collapsed}) = \varnothing, \end{aligned}$$

This *refusal completion table* is the single source from which the rest of the book's descriptions of the lifecycle are derived: the operational rules of Chapter 9, the lifecycle step function of Chapter 17, which is the table with the data carried by each state added, and the session type of Chapter 15, which is the table read as a protocol. Every row for a live state contains refuse; every terminal row is empty. Those two facts are, in miniature, the statements that refusal is always available and that terminal dispositions are final. The verify entry has two outcomes, to verified and to verificationFailed. A policy may extend the failed row, for instance with a fresh verification attempt or replacement evidence, but nothing is added to it implicitly.

1.4 Records and the step function

Every disposition, and every discharge of an external obligation, leaves a record in the history. The records are

$$\begin{gathered} \mathsf{Pop}(c, q), \quad \mathsf{Compensates}(a, c, q, d), \quad \mathsf{Bind}(a, c, b), \quad \mathsf{Transform}(c, f, b, v), \\ \mathsf{Verify}(a, c, v, w), \quad \mathsf{VerificationFailed}(a, c, v, \varphi), \\ \mathsf{Refuse}(a, c, r), \quad \mathsf{Collapse}(a, c, k, w, o), \quad \mathsf{Discharged}(a, c, \omega, \varphi), \quad \mathsf{TimedOut}(a, c, \omega, t, \varphi), \end{gathered}$$

where a names a principal, q a request, b and v values, f a transformation, w a witness, k an observation rule, and o an observation.

A verifier g applied to a commitment and value returns one of two results: $\mathsf{ok}(w)$, a witness, or $\mathsf{fail}(\varphi)$, a *failure certificate*. Certificates have a fixed shape,

$$\varphi = \langle \kappa, g, c, h(v), \mathsf{outcome}, \sigma \rangle,$$

naming the verifier's semantic version κ, the issuer g, the commitment c, a hash $h(v)$ of the value or claim the certificate is about, the outcome it attests, and a signature σ over all of these. The same shape serves for external evidence of discharge, below, with the obligation in place of the value. A negative verification is recorded by its own record, $\mathsf{VerificationFailed}(a, c, v, \varphi)$, carrying the certificate. Refusal reasons r are always written by the refusing principal; there is no reserved reason, because a failed verification is not a refusal.

The *subject* of a record is the commitment whose state it changes. For a compensation record it is the new successor c; the predecessor d is referred to but not changed. In every other record it is the commitment named first after the principal, or first overall for Pop and Transform. For a history H, the *projection* $H|_c$ is the subsequence of records whose subject is c, in ledger order.

Two predicates on evidence are fixed: $\mathrm{valid}(w, c, v)$, which says w is a correct witness for c having value v, and

$$\mathsf{checkCert}(\varphi, c, x, \mathsf{outcome}),$$

which says that φ is well formed, that its commitment field is c, its claim field is $h(x)$, its outcome field is the one named, and its signature verifies for the issuer and version it names. The check reads the certificate; it never reruns the verifier. Rerunning may be impossible, expensive, nondeterministic, or altered by a later version, and replay must remain deterministic when the original verifier is gone.

Definition 1.3 (Lifecycle step)**.** *The partial function δ takes a state, or $\bot$ for "not yet introduced", and a record, and returns the resulting state:*

$$
\begin{aligned}
\delta(\bot, \mathsf{Pop}(c,q)) &= \mathsf{open}(q), \\
\delta(\bot, \mathsf{Compensates}(a,c,q,d)) &= \mathsf{open}(q), \\
\delta(\mathsf{open}(q), \mathsf{Bind}(a,c,b)) &= \mathsf{bound}(q,b), \\
\delta(\mathsf{bound}(q,b), \mathsf{Transform}(c,f,b,v)) &= \mathsf{transformed}(b,v), \\
\delta(\mathsf{transformed}(b,v), \mathsf{Verify}(a,c,v,w)) &= \mathsf{verified}(v,w) && \text{if } \mathrm{valid}(w,c,v), \\
\delta(d, \mathsf{Refuse}(a,c,r)) &= \mathsf{refused}(r) && \text{if } \mathsf{live}(d), \\
\delta(\mathsf{transformed}(b,v), \mathsf{VF}(a,c,v,\varphi)) &= \mathsf{vFailed}(v,\varphi) && \text{if } \mathsf{checkCert}(\varphi,c,v,\mathsf{fail}), \\
\delta(d, \mathsf{Discharged}(a,c,\omega,\varphi)) &= d && \text{if } \mathsf{live}(d) \text{ and} \\
& && \quad \mathsf{checkCert}(\varphi,c,\omega,\mathsf{discharged}), \\
\delta(d, \mathsf{TimedOut}(a,c,\omega,t,\varphi)) &= d && \text{if } \mathsf{live}(d) \text{ and} \\
& && \quad \mathsf{checkCert}(\varphi,c,\langle\omega,t\rangle,\mathsf{expired}), \\
\delta(\mathsf{verified}(v,w), \mathsf{Collapse}(a,c,k,w,o)) &= \mathsf{collapsed}(o) && \text{if } o = \mathsf{observe}_k(v),
\end{aligned}
$$

where VF *abbreviates the record* VerificationFailed *and* vFailed *the state* verificationFailed*, and δ is undefined in every other case. Repeated variables must match: the b of a transformation record must be the b held in the bound state, the v of a verification record the v produced by transformation, and the w of a collapse record the witness held in the verified state. Write δ^* for the left fold of δ over a sequence from $\bot$.*

The step function is the refusal completion table with the data carried by each state checked against the data in each record. Discharge and timeout records are the two clauses that leave the lifecycle state unchanged. A discharge records that principal a discharged an external obligation ω that c was waiting on, with authenticated evidence φ. A timeout records, with evidence from a clock authority a, that ω was *not* discharged within the declared interval t. Neither ends the commitment; both are consumed and checked by the fold, not skipped. Formally δ also carries, beside each state, the set of obligations already settled for c, by discharge or by timeout, and rejects a second settlement of the same obligation; since that set never affects any other clause, it is suppressed from the notation. Note the verification-failure clause: it is admissible only from transformed, only with an authentic certificate for that commitment and value, and it leads to a live state, not a terminal one. Refusal is admissible from every live state, verificationFailed included, and only by explicit act.

Hypothesis 1.4 (Stability)**.** *The predicates* valid *and* checkCert *and the maps* $\mathsf{observe}_k$ *are deterministic and fixed for the duration of an execution, relative to the verifier versions recorded in witnesses and certificates.*

Stability is what lets a verification carried out at one moment count as evidence at a later moment, and what lets replay re-check a record long after it was appended. Versions matter

because a verifier upgraded mid-execution changes what valid means; recording κ keeps the question "valid under which verifier?" answerable.

Hypothesis 1.5 (Verifier soundness)**.** *If* $g(c, v) \Downarrow \mathsf{ok}(w)$ *then* $\mathrm{valid}(w, c, v)$*, and if* $g(c, v) \Downarrow \mathsf{fail}(\varphi)$ *then* $\mathsf{checkCert}(\varphi, c, v, \mathsf{fail})$*. Moreover* checkCert *holds only for certificates actually issued by the issuer named in* φ*, at the version named, on the commitment and claim named.*

The first half is a requirement on verifiers. The second is an authentication assumption, met in practice by signing certificates with a key held by the verifier.

1.5 A first example

Consider a request q that introduces a fresh commitment c. A value b is bound to it by principal a, a function f transforms b into v, a verifier a' produces witness w, and a'' collapses the result under observation rule k. The successful history is

$$H = \langle \mathsf{Pop}(c, q),\ \mathsf{Bind}(a, c, b),\ \mathsf{Transform}(c, f, b, v),\\ \mathsf{Verify}(a', c, v, w),\ \mathsf{Collapse}(a'', c, k, w, o)\rangle.$$

The identity c threads through the record. It is not replaced by b or v. The witness is indexed by c and v, preventing evidence produced for another commitment from being replayed here.

If verification instead rejects v, the history continues

$$\mathsf{VerificationFailed}(a', c, v, \varphi)\,;\ \mathsf{Refuse}(h, c, r),$$

where φ is the verifier's authenticated failure certificate and h is a principal with authority to refuse. The verifier changed the evidentiary state of the commitment; the refusal is a separate act by someone entitled to dispose of it. This is not a truncated successful history. It is a complete refused history, and it retains the evidence of failure beside the decision that followed it. Any later reconsideration must introduce a successor commitment c' and record that c' compensates for the practical matter associated with c.

Chapter 2

Disposition Worlds

A formal semantics becomes usable when a reader can build small models of it by hand, ask whether statements are true in them, and find the smallest model in which a plausible statement fails. This chapter introduces the finite models used throughout the book, together with three instruments for working with them: countermodels, history twins, and causal shuffles. They are elementary enough to use before any metatheory is available, and later chapters show that they were instantiating the central theorems all along. A computation is a finite causal world; a ledger is one serialization of that world; and a collapse rule declares what an observer retains.

2.1 Worlds and configurations

Definition 2.1 (Disposition world)**.** *A* disposition world *is a tuple*

$$W = \langle E, \prec, \#, \lambda, \mathcal{P}, \mathcal{K} \rangle$$

where E is a finite set of events; $\prec$ is a strict partial order on E (causal precedence); $\#$ is a symmetric, irreflexive relation on E (incompatibility) that is inherited upward, so that $e \# e' \prec e''$ implies $e \# e''$; λ labels each event with a ledger record; $\mathcal{P}$ is an authority policy, a set of triples (a, κ, c) stating that principal a may perform acts of kind κ on commitment c; and $\mathcal{K}$ is a finite set of observation rules. The commitments of W are the subjects of its labels.

A *configuration* of W is a set $x \subseteq E$ that is downward closed under $\prec$ and contains no two incompatible events. It is one possible state of the computation: the events that have happened. A *linearization* of x is a listing of its events consistent with $\prec$, read as a ledger through λ. A *pointed world* (W, x) fixes a current configuration.

Definition 2.2 (Admissible world)**.** *W is* admissible *when*

(W1) any two events whose records are dependent, in the sense of Definition 20.9, are either $\prec$-comparable or incompatible;

(W2) every linearization of every configuration has a defined replay;

(W3) for every event labelled with an attributed record, the required authority of Chapter 9 is in $\mathcal{P}$.

Condition (W1) says the world orders everything that replay cares about. Condition (W2) says every possible state of the world is a legal disposition state. Condition (W3) says nothing in the world was done without entitlement.

Worlds of this kind are prime event structures, in the sense of Nielsen, Plotkin, and Winskel, with events labelled by records and an authority policy attached. The book's first comparative claim is that this is not an accident, and this chapter is where the resemblance first becomes concrete.

A world with empty # has a single maximal configuration and describes one execution. Every execution produces one:

Proposition 2.3 (Executions are worlds)**.** *Let H be the history of an execution from the empty configuration. The world whose events are the records of H, ordered by the dependence order $\prec_H$ of Definition 20.9, with empty incompatibility and the execution's policy, is admissible.*

Proof. (W1) holds because dependent records are $\prec_H$-ordered by definition. For (W2), let x be a downset of $\prec_H$. Some linear extension of $\prec_H$ lists x first; it is trace equivalent to H, so by Proposition 20.10 its replay is defined, and replay of a prefix of a defined fold is defined. Any other linearization of x is trace equivalent to that prefix (Theorem 2.9). (W3) is Theorem 9.9. □

What an ordinary event structure lacks

A disposition world has a plain labelled event structure underneath it. Write it as

$$\mathcal{E}_D = \langle E, \leq, \#, \lambda, \eta \rangle,$$

where $\langle E, \leq, \#, \lambda \rangle$ is a labelled prime event structure whose labels $\lambda(e)$ record only the kind of act and its subject, and η carries the disposition annotations: principal, reason, witness, certificate, verifier version, and observation rule. The underlying structure determines configurations and linearizations, and with them everything in the Causal Shuffle theorem below. What it does not determine is admissibility: conditions (W2) and (W3) consult η through valid, $\mathsf{checkCert}$, $\mathsf{observe}_k$, and the policy.

This makes the first comparative claim precise. It is settled in the positive if (W2) and (W3) can be expressed as structural constraints on a labelled event structure over a finite alphabet, in which case the calculus is an event-structure discipline with rich labels. It is settled in the negative if some admissibility condition depends irreducibly on evidence predicates over unbounded data, in which case η is the residue the claim predicts.

2.2 A small language

Formulas are built from atomic statements about a pointed world, the Boolean connectives, quantification over the finitely many commitments, principals, and values of W, and one modality.

- $\mathsf{Live}(c)$, $\mathsf{Refused}(c)$, $\mathsf{Collapsed}(c)$, and the other state predicates hold at (W, x) when replay of a linearization of x puts c in the named state.
- $\mathsf{Occurred}(\rho)$ holds when some event of x is labelled ρ.

- $\mathsf{Authorized}(a, \kappa, c)$ holds when $(a, \kappa, c) \in \mathcal{P}$.
- $\mathsf{Valid}(w, c, v)$ is the validity predicate.
- $\Diamond\varphi$ holds at (W, x) when φ holds at (W, y) for some configuration $y \supseteq x$. $\Box$ is its dual.

The state predicates are well defined only because every linearization of x gives the same replay. That is the Causal Shuffle theorem below, so the language rests on it from its first clause.

Proposition 2.4 (Modal terminal exclusivity)**.** *In every admissible pointed world,*

$$(W, x) \models \mathsf{Refused}(c) \to \neg\Diamond\mathsf{Collapsed}(c).$$

Proof. Suppose c is refused at x and $y \supseteq x$. The refusal event of c is in y. In any linearization of y, the projection onto c contains a refusal record, and since δ is undefined on every record whose incoming state is terminal, that refusal is the last record of the projection and the fold yields $\mathsf{refused}$. So c is not collapsed at y. $\Box$

2.3 Countermodels

The most useful operation on worlds is the search for the smallest one that falsifies a proposed statement:

$$\mathsf{countermodel}(\varphi, n) = \text{a minimal admissible } W \text{ with at most } n \text{ events and } W \not\models \varphi.$$

Minimality is lexicographic: fewest commitments, then fewest events, then fewest principals, then least causal depth. Small countermodels teach more than arbitrary failing cases, because every event in a minimal countermodel is doing work.

Two examples, found by the laboratory of Section 2.9 under a policy in which a_1 may bind and a_2 may verify:

- The conjecture "if c is verified, the principal who bound c is authorized to verify it" fails in a four-event world, $\mathsf{Pop}(c)\,;\mathsf{Bind}(a_1, c)\,;\mathsf{Transform}(c)\,;\mathsf{Verify}(a_2, c)$. The conjecture was underspecified: authority attaches to acts, not to commitments, and different acts on one commitment may belong to different principals.
- The conjecture "a refused commitment was never verified" fails in a five-event world in which a reviewer refuses after a successful verification. Refusal is available from $\mathsf{verified}$ as from every live state, and a valid witness does not oblige anyone to accept.
- The conjecture $\mathsf{Live}(c) \to \Diamond\mathsf{Collapsed}(c)$, that every live commitment can still collapse, fails in a four-event world, $\mathsf{Pop}(c)\,;\mathsf{Bind}(a_1, c)\,;\mathsf{Transform}(c)\,;\mathsf{VF}(a_2, c)$. A commitment whose verification failed is live, but no continuation leads to collapse; the only way out is refusal. This countermodel appeared when the laboratory was rerun after the $\mathsf{verificationFailed}$ state was introduced, and it is correct: it is the formal content of the decision that a negative verification does not dispose, and that nothing but an authorized refusal disposes after it.

Conversely, no countermodel with at most six events exists for Proposition 2.4, nor for the weaker reachability statement $\mathsf{Live}(c) \to \Diamond\mathsf{Terminal}(c)$ under that policy.

Remark 2.5 (Horizons). A bounded search for $\Diamond\varphi$ must look far enough ahead. A search that extends worlds only to the global bound will report $\mathsf{Pop}(c_1)\,;\mathsf{Bind}(a_1, c_1)\,;\mathsf{Pop}(c_2)$ as a countermodel to $\mathsf{Live}(c) \to \Diamond\mathsf{Terminal}(c)$ at bound six, simply because the remaining budget is too small for c_1 to finish. The laboratory therefore extends each world by the longest lifecycle path, four events, beyond its own length. The general moral is that a bounded countermodel to a modal statement is evidence only relative to its horizon.

The same care applies to what the positive result means. The modality $\Diamond$ is existential, so the absence of a countermodel to $\mathsf{Live}(c) \to \Diamond\mathsf{Terminal}(c)$ is a *reachability* result: from every live state some continuation of at most four records reaches a terminal state, and when any principal holds refusal authority, one record suffices. It is not the liveness statement that every execution eventually disposes of c. Unrelated events may postpone c indefinitely and external obligations may never be answered; eventual disposition is a conditional temporal theorem, stated with its hypotheses in Chapter 23.

2.4 History twins and collapse lenses

A *lens* is an observation rule k applied to a pointed world, yielding some value $k(W, x)$. Two pointed worlds are equivalent under k, written $(W, x) \approx_k (W', x')$, when their lens values agree.

Definition 2.6 (History twins)**.** *Pointed worlds are* twins *for lenses* (k_1, k_2) *when they are equivalent under* k_1 *and not under* k_2*.*

The laboratory finds three canonical twins.

1. *Same output, different refusals.* A single commitment that is bound, transformed, verified, and collapsed to o; and a world in which the same happens while a second commitment is opened and refused. A lens reporting the observed values sees $\{o\}$ in both. A lens reporting refusals distinguishes them.

2. *Same output, different authority.* The same five-event history twice, once collapsed by a_1 and once by a_2. The value lens cannot tell them apart; a lens reporting who performed the decisive act can.

3. *Same state, different refusal.* The worlds

   $$\mathsf{Pop}(c)\,;\mathsf{Refuse}(a_1, c, r) \quad \text{and} \quad \mathsf{Pop}(c)\,;\mathsf{Bind}(a_1, c)\,;\mathsf{Transform}(c)\,;\mathsf{VF}(a_2, c)\,;\mathsf{Refuse}(a_1, c, r).$$

   Both leave c refused. One refusal was a decision before any work was done; the other was a decision taken after a certified verification failure, and the ledger keeps the failure beside the decision. This is the operational content of indexing the refusal type by the lifecycle stage, $\mathsf{Refusal}\langle c, A, s\rangle$, in Chapter 14.

Two worlds are never simply equivalent. They are equivalent under a declared lens.

Definition 2.7 (Lens refinement)**.** *Lens* k_1 refines k_2*, written* $k_1 \preceq k_2$*, when* $k_2 = h \circ k_1$ *for some function* h*: everything* k_2 *reports can be computed from what* k_1 *reports.*

Proposition 2.8. *Refinement is a preorder, and if* $k_1 \preceq k_2$ *then* $(W, x) \approx_{k_1} (W', x')$ *implies* $(W, x) \approx_{k_2} (W', x')$*.*

Proof. Reflexivity takes h to be the identity and transitivity composes the two functions. If k_1 values agree, applying h to both gives equal k_2 values. □

Lenses identified up to mutual refinement correspond to the equivalence relations they induce on pointed worlds, and equivalence relations on a set form a complete lattice under inclusion. The collapse rules of the calculus therefore form a structured spectrum rather than an unrelated collection. Whether that spectrum coincides with van Glabbeek's for the classical equivalences, when those are recast as lenses, is the question of Chapter 24.

2.5 Causal shuffles

The third instrument enumerates every ledger that could have recorded one world: all linearizations of a configuration.

Theorem 2.9 (Causal shuffle)**.** *In a world satisfying (W1), all linearizations of a configuration have the same replay: either all are undefined, or all are defined and equal. In particular this holds in every admissible world.*

Proof. Any two linear extensions of a finite partial order are connected by a sequence of exchanges of adjacent elements that are incomparable in the order: move the element that comes first in the target order to the front by successive adjacent exchanges, each of which passes an element that the target places later and which therefore cannot precede it in the order, and recurse on the rest. Two incomparable events in a configuration are not incompatible, so by (W1) their records are independent. By Proposition 20.10, exchanging adjacent independent records does not change replay, and does not change whether it is defined. The proof uses only (W1), which is why it can be invoked in establishing (W2) for Proposition 2.3. □

The theorem identifies exactly when the sequence numbers a sequencer assigns are incidental. Every order among independent records is one serialization of the same world. Order is semantically necessary only between dependent records, and (W1) requires the world to state it. For the finite class $\mathcal{H}_{\leq 7}$ of Section 2.9, the laboratory confirms the theorem instance by instance: each of the 4,358,436 linearizations of the 56,397 ledgers in the class replays to the same state as the ledger it came from.

2.6 The witness mixer

Let c_1 and c_2 both be $\mathsf{transformed}(b, v)$, with witnesses w_1 produced for (c_1, v) and w_2 produced for (c_2, v). Try verifying c_1 with w_2. A check that asked only whether the witness vouches for the value v would accept, since the values coincide. The step function refuses, because it asks for $\mathrm{valid}(w_2, c_1, v)$.

That refusal depends on the verifier, not only on the calculus. Indexing forces the question to be asked about the right commitment, but the predicate must actually distinguish commitments for the answer to differ.

Hypothesis 2.10 (Index sensitivity)**.** $\mathrm{valid}(w, c, v) \wedge \mathrm{valid}(w, c', v)$ *implies* $c = c'$.

Under this hypothesis evidence is non-transferable: a witness valid for one commitment is valid for no other, and the witness ancestry of Corollary 18.3 cannot be satisfied by borrowed evidence. The hypothesis is a requirement on verifiers, typically met by signing or hashing the commitment identity into the witness, and it is stated here because nothing in the operational rules can supply it.

2.7 Perturbations

The laboratory's third mode takes an admissible world and changes exactly one thing. On the five-event successful history, every perturbation but one is caught, and each is caught by a specific clause. A sixth row comes from a third base, a commitment refused after a certified verification failure, whose certificate is replaced by a forged one:

Perturbation	Detected by
unauthorized binder	authority (W3)
foreign witness in verification	δ: valid(w, c, v) fails
collapse citing a different witness	δ: witness mismatch
exchange of two dependent records	δ: wrong incoming state
reuse of a commitment identity	δ: identity not fresh
forged failure certificate	δ: checkCert fails
deletion of an interior record	δ: wrong incoming state, *usually*
deletion of the final record	*not detected*

The last row is not a flaw in the checker. A ledger with its final record removed is a prefix of an admissible ledger, and every prefix of an admissible ledger is admissible, since the world it describes is simply an earlier configuration. Replay certifies that what is present could have happened. It cannot certify that nothing is missing from the end.

This is an impossibility, not an oversight, and Chapter 20 states it as the Prefix Indistinguishability theorem together with the remedy: a signed checkpoint retained outside the ledger. Completeness itself requires a witness outside the history whose completeness is in question.

The qualification in the interior row is a finding of the second base. Deleting the transformation from

$$\mathsf{Pop}(c)\,;\mathsf{Bind}(a_1, c)\,;\mathsf{Transform}(c)\,;\mathsf{Refuse}(a_1, c, r)$$

leaves $\mathsf{Pop}(c)\,;\mathsf{Bind}(a_1, c)\,;\mathsf{Refuse}(a_1, c, r)$, which is admissible, because explicit refusal is legal from the bound state as well. The same generosity that makes refusal available at every live stage makes some interior deletions invisible to replay. Any deletion whose remainder is still a legal path escapes detection, and the same anchor is the remedy: a hash chain detects every modification, not only truncation.

2.8 Disposition control

The instruments so far evaluate statements in given worlds. A different question asks which decisions determine a disposition: what must be fixed so that a commitment is guaranteed to end one way.

Definition 2.11 (Interventions, forcing, influence)**.** *An* intervention K *is a set of constraints on decision points, each removing some records from the enabled set wherever they would occur:*

for instance "verifications of c succeed" or "principal a never refuses c". A completion *under K is a maximal admissible ledger all of whose steps respect K. Then*

- $\mathsf{force}(K, d)$ *holds when* every *completion under K leaves c in disposition d;*
- $\mathsf{influence}_\mu(K, d) = \Pr_\mu\big[$*a completion under K leaves c in d*$\big]$*, for an explicitly declared distribution μ over schedules, initial states, or worlds.*

The two are different kinds of claim. Forcing is a universal statement over completions and is proved or refuted by exhaustion. Influence is a probability and depends on μ; a high influence is not a guarantee. Without a declared μ, a figure of this kind is not a probability at all: it can only be a coverage figure or a reachability count, and should be reported as one. The distinction is taken from recent work on control of Boolean network models of biological systems, where controlling a feedback vertex set together with the source nodes carries a structural guarantee of reaching a target attractor, while smaller control sets found through network modularity are evaluated by *percent coverage*: the proportion of the target outcome that fixing the control set is shown to reproduce. In that study's biological models, one variant reached about 87% average coverage with control sets slightly larger than the minimum feedback vertex set, and another, choosing one high-ranked feedback vertex per module, reached about 77% with sets about 63% of that size [24]. Coverage measures partial achievement of a target, and influence as defined here is a frequency over schedules, so the two quantities differ. The lesson carries over all the same: a measure of partial success, however high, is not a guarantee.

For a single commitment under the laboratory's policy, with μ the uniform scheduler that picks among enabled records with equal probability:

- With no intervention, the influence toward collapse is 1/72 and toward refusal 71/72. The asymmetry reflects only that refusal options outnumber advancing ones at every stage under a uniform scheduler, not anything about real systems.
- The unique minimal forcing set for *collapse* has three elements: verifications succeed, and neither principal refuses.
- The unique minimal forcing set for *refusal* has one element: verifications fail.
- The best two-element set that does not force collapse, "neither principal refuses", reaches influence 1/2: every remaining completion turns on the verifier.

Forcing refusal is cheap and forcing collapse is expensive, because refusal is available everywhere and collapse only at the end of one path. That asymmetry is structural. It is the formal shadow of the book's insistence that refusal be available from every live state.

Proposition 2.12 (No cycles, one re-entry)**.** *The lifecycle is acyclic, and each commitment's projection contains at most one opening, four lifecycle steps, and one discharge per obligation. The only rule enabled by a terminal state is compensation, and it creates a new commitment rather than re-entering the old one. Hence in this calculus an infinite execution is never a cycle in any commitment's lifecycle; it is an unbounded supply of new commitments, arriving through replication or through chains of compensation.*

Proof. Acyclicity and the bound on lifecycle steps are properties of δ. Discharges are bounded by the obligation set. Compensation is the only rule with a terminality premise, and its

subject is fresh. An infinite execution therefore introduces infinitely many commitments, and introductions come only from pop, whose instances beyond the finitely many in the program text come from replication, and from compensation. □

In the Boolean-network setting, a feedback vertex set matters because cycles are what sustain undesired attractors. Here there are no cycles to break in the basic calculus, and a feedback-set analysis would be ornamental. What plays the corresponding role is the set of reentry points: replication, and the authority to compensate. Removing compensation authority leaves replication as the only source of unbounded work. If the calculus is extended with reopening, retries, delegation cycles, or mutually dependent verification, genuine cycles appear, and a theorem of the feedback-set kind becomes available: an intervention that meets every live dependency cycle removes cyclic obstructions to disposition. It would still not guarantee eventual disposition, which needs fairness and responsive dependencies as well (Chapter 23).

2.9 The laboratory

The examples in this chapter were produced by a small executable reference, `disposition_lab.py`, which implements the records, the step function, and replay as the book defines them, and enumerates every admissible ledger up to a bound by running a maximally nondeterministic process under a fixed policy. Its results are bounded propositions about a named finite class, not theorems, and they are versioned with the grammar they were computed for.

Let $\mathcal{H}_{\leq 7}$ be the class of admissible ledgers of at most seven records over two commitment identities, two principals, one value, one transformation, one observation rule, one external obligation per commitment with one declared deadline, one exclusive capability (refusal of c_1) that may be granted and delegated, and the policy stated in the source. For version 5 of the laboratory, whose grammar is the book's current record grammar, including the verificationFailed state, authenticated certificates, discharge and timeout records, and grant and delegation records with capability ownership in the replay map, $|\mathcal{H}_{\leq 7}| = 56{,}397$. The laboratory reports each check as a named property with its class, search depth, normalization, expected result, and, when it fails, a minimized counterexample, so that it serves as an executable index of the proof spine. Every property below holds on every $H \in \mathcal{H}_{\leq 7}$:

Property	Statement checked
P-ReplaySound	δ^* of each projection equals replay's map (the path lemma)
P-TerminalExclusive	at most one terminal record per commitment, and it is last
P-CertificateAuthentic	every verification failure checks against its state and a verifier
P-FailedVerificationLive	a verification failure leaves its commitment live
P-NoVerifierDisposition	no verification record yields a terminal state
P-DelegationConservative	one owner per capability; delegation and exercise only by it
P-TraceInvariant	all 4,358,436 linearizations replay to the state of their ledger
P-WorldAdmissible	every configuration has a defined replay

In addition, for each of the 47,889 pointed worlds in which some commitment is refused, no admissible extension within the horizon collapses it. The companion module `disposition_parser.py` checks the parser properties of Chapter 6, and `disposition_es.py` checks the event-structure properties of Chapter 21. One property named in the proof spine, P-RightsAgree, needs the process and typed layers, which the laboratory does not model; it is

proved in Chapter 15 and not checked here. P-DelegationConservative, listed as unchecked in earlier versions, is checked from version 5 at the ledger level: it confirms that replay admits only owner-initiated delegations and exercises, not the typing argument of Chapter 16.

Earlier versions checked earlier grammars, and their figures apply to those grammars only: version 1 (1,230 ledgers), version 2 with certificates and discharge (6,305), version 3 with timeout and automatic refusal on failed verification (14,608), and version 4 with the verificationFailed state and no capabilities (15,167, with 372,961 linearizations). The larger count in version 5 comes from the grant and delegation records, which interleave freely with the records of c_2. The countermodels, twins, and control figures are unchanged across versions, with one deliberate exception: the countermodel to $\mathsf{Live}(c) \to \Diamond\mathsf{Collapsed}(c)$, which exists only once a failed verification leaves its commitment live. The laboratory's value is not in the counts. It is in finding the smallest world where an idea goes wrong, which is how the horizon remark, the interior-deletion finding, the forged-certificate row, and that countermodel were found.

Exercises

1. Construct the smallest admissible world in which a commitment is compensated. How many events does it need, and why can none be removed?

2. Find twins for the lenses "set of commitments in a terminal state" and "set of observations".

3. Give a world violating (W1) in which two linearizations of the same configuration replay differently.

4. Define a lens that distinguishes twin 3 above but not twin 1, and place it in the refinement order relative to the value lens and the refusal lens.

5. Show that without Hypothesis 2.10 there is an admissible world in which c_1 is verified with a witness that was produced for c_2.

Chapter 3

The Kernel and the Disposition Layer

The disposition calculus uses the names of the Spherepop kernel's operators. This chapter decides whether that is a formalization of the kernel or a related calculus built above it, by giving an explicit translation

$$[\![-]\!] : \text{disposition records} \longrightarrow \text{kernel histories}$$

and proving two things about it: that every disposition step translates to an admissible kernel step (forward simulation), and that every kernel step of translated shape whose guard holds comes from a disposition step (backward adequacy). The records and the step function it uses were defined in Section 1.4; the operational rules the guards refer to are stated in Chapter 9, and the corollary relating the two uses consistency, established in Chapter 20. Until both directions are proved, the relation between the layers is a conjecture, and it should be read that way.

3.1 The kernel, restated

A kernel history is an append-only sequence of events. Four causal operators act on histories.

- $\mathsf{Pop}(x)$ commits to the option x among those currently available, removing it from what remains possible.
- $\mathsf{Refuse}_r(X)$ documents that the options in X are inadmissible, with reason r, without removing them from the record of what was considered.
- $\mathsf{Bind}(H_1, H_2)$ couples two histories so that each constrains the other's continuation, without identifying them.
- $\mathsf{Collapse}_k(H)$ observes a history under a rule k, projecting it onto the quotient k induces.

Three structural rules build the terms these operators act on: sequence $H_1; H_2$, tensor $H_1 \otimes H_2$, which forms a joint context from two histories so that an operator can condition one on the other, and branch, which divides one history into two continuations sharing a prefix. A separate annotation channel carries metadata that participates in no causal reduction.

One feature of the kernel matters more than any other here. Applying $\mathsf{Collapse}_k$ computes an observation. It does not by itself make that observation a commitment: a history can be observed under many rules, and none of those observations constrains what happens next unless something commits to it. The corpus's operating-system specification states this as "collapse is rendering" and keeps rendered views out of the authoritative log. The translation below adopts that reading. Under the alternative reading, in which every kernel collapse is itself logged, one step of the translation is absorbed and nothing else changes; the paragraph on collapse in Section 3.4 says which.

3.2 Kernel histories and admissibility

To state a translation precisely, the kernel needs a semantics of its own, and it should be the most permissive one consistent with the four operators. Fix a universe $\mathcal{X}$ of options. A *kernel history* K is a finite sequence of events

$$\mathsf{Pop}(x), \qquad \mathsf{Refuse}_r(X), \qquad \mathsf{Bind}(x, y),$$

for options $x, y \in \mathcal{X}$, sets $X \subseteq \mathcal{X}$, and reasons r. Write $\mathsf{Popped}(K)$ for the options popped in K and $\mathsf{Refused}(K)$ for the union of the refused sets. An event is *kernel admissible* after K when:

- $\mathsf{Pop}(x)$: $x \notin \mathsf{Popped}(K) \cup \mathsf{Refused}(K)$, so an option can be committed to at most once, and never after it has been documented as inadmissible;
- $\mathsf{Refuse}_r(X)$: always, since documenting inadmissibility removes nothing;
- $\mathsf{Bind}(x, y)$: $x, y \in \mathsf{Popped}(K)$, so only things that exist can be coupled.

A kernel history is admissible when each event is admissible after its prefix. The kernel's derived choice, $\mathsf{Choice}(x, X) = \mathsf{Pop}(x) \parallel \mathsf{Refuse}(X \setminus \{x\})$, commits to one option and documents the rest as inadmissible. Kernel $\mathsf{Collapse}_k$ is a function on histories, a rendering, and not an event.

This semantics enforces very little, and that is deliberate. It says nothing about the order in which options may be popped, nothing about what data they carry, and nothing about who may pop them. Whatever the disposition calculus requires beyond freshness and the permanence of refusal must therefore be supplied by the translation, and the question of this chapter is exactly how much that is.

3.3 Continuation spaces and guards

The options of the translation are the records themselves. A record popped in the kernel is a commitment to that record having occurred.

Definition 3.1 (Continuation space)**.** *For a ledger H and commitment c, the* admissible continuation space *is*

$$\Omega_c(H) = \{\tau \mid \textit{every record of } \tau \textit{ has subject } c \textit{ and } \delta^*(H|_c \cdot \tau) \textit{ is defined}\},$$

the set of futures that c's own lifecycle still admits. Its one-step part, the records that could come next, is $\Omega_c^1(H) = \{\rho \mid \langle\rho\rangle \in \Omega_c(H)\}$. Write $\mathsf{Opt}_c(H)$ for the set of all records with subject c not yet popped.

For a live commitment $\Omega_c(H)$ contains the continuations its state, the data it carries, and the evidence predicates allow. For a terminal commitment it contains only the empty continuation.

Definition 3.2 (Guard)**.** *The* guard *of a record ρ at ledger H, written $G_\rho(H)$, is the conjunction of the premises of the operational rule that appends ρ, read against the state that replay of H reconstructs: the lifecycle and data conditions ($\rho \in \Omega^1_{\mathrm{subj}(\rho)}(H)$), the authority premise, the evaluation or verification premise where there is one ($f\, b \Downarrow v$, $g(c, v) \Downarrow \mathsf{ok}(w)$ or $\mathsf{fail}(\varphi)$), and for compensation the terminality of the predecessor.*

The guard is a statement in the metalanguage. It is not a kernel operator, and the translation does not emit an event for it. A step whose guard holds is translated; a step whose guard fails is simply not a disposition step. If an implementation wishes to record a failed attempt, it may do so as a kernel refusal of the proposed option or on the annotation channel, but nothing in what follows depends on that choice.

3.4 The translation

Definition 3.3 (Record translation)**.** *For a record ρ with subject c whose guard $G_\rho(H)$ holds, let ℓ_c be the last record of $H|_c$, and define $[\![\rho]\!]_H$ by*

$$
\begin{aligned}
[\![\mathsf{Pop}(c, q)]\!]_H &= \mathsf{Pop}(\rho), \\
[\![\mathsf{Compensates}(a, c, q, d)]\!]_H &= \mathsf{Pop}(\rho); \mathsf{Bind}(\rho, \ell_d) \quad \textit{over } H_c \otimes H_d, \\
[\![\rho]\!]_H &= \mathsf{Pop}(\rho); \mathsf{Bind}(\rho, \ell_c) \quad \textit{for every other non-terminal } \rho, \\
[\![\mathsf{Refuse}(a, c, \varrho)]\!]_H &= \mathsf{Pop}(\rho) \parallel \mathsf{Refuse}_{\langle a, \varrho \rangle}(\mathsf{Opt}_c(H) \setminus \{\rho\});\ \mathsf{Bind}(\rho, \ell_c), \\
[\![\mathsf{Collapse}(a, c, k, w, o)]\!]_H &= \mathsf{Pop}(\rho) \parallel \mathsf{Refuse}(\mathsf{Opt}_c(H) \setminus \{\rho\});\ \mathsf{Bind}(\rho, \ell_c), \quad o = \mathsf{Collapse}_k(v).
\end{aligned}
$$

*The other non-terminal records are the advancing records (*Bind*,* Transform*,* Verify*) and the discharge and timeout records. The translation of a ledger is $[\![\varepsilon]\!] = \varepsilon$ and $[\![H \cdot \rho]\!] = [\![H]\!]; [\![\rho]\!]_H$, defined when every guard holds at its prefix. Conversely, the* decoding *of a kernel history lists its popped records in order.*

The kernel history component H_c of a commitment is the chain of its popped records linked by Bind: each record of c is coupled to the one before it, without being identified with it. The clauses fall into four groups.

Openings. Introducing a commitment pops its opening record. Freshness needs nothing extra: a popped option can never be popped again. Compensation also binds the new opening to the predecessor's last record, which needs the tensor rule to form the joint context of two histories, the same requirement that waiting on a child process and mounting a namespace place on the operating-system specification. Bind is the right operator because it couples without identifying, the property developed at length for the kernel's binding in [36]: the successor depends on the predecessor and is a different commitment.

Advancing, discharge, and timeout. Binding, transformation, verification, discharge, and timeout all have the same shape, $\mathsf{Bind} \circ \mathsf{Pop}$ onto the commitment's history. They differ

in payload and in guard. Nothing about transformation or verification requires a kernel operator the other two do not; their distinctive content is in the conditions under which they are admissible.

Terminal steps. Refusal and collapse are both instances of the kernel's derived choice: one option is popped and every other option of c is documented as inadmissible. What distinguishes them is what is popped. A refusal pops an attributed, reasoned decision, and the refused set carries that principal and reason. A collapse pops a committed observation $o = \mathsf{Collapse}_k(v)$: kernel collapse renders an observation, and the pop makes it a commitment. Disposition refusal is therefore kernel refusal of the *whole remaining option space* of c, which is what distinguishes it in scope from kernel refusal of a single option; and disposition collapse is $\mathsf{Pop} \circ \mathsf{Collapse}_k$, a committed observation, rather than kernel collapse alone.

Where the discipline lives. Because the terminal translations refuse every remaining option of c, the kernel's own admissibility enforces finality: after either terminal step, no further record of c can be popped. Freshness is likewise enforced by the kernel. Everything else, the order of the lifecycle, the matching of data between records, the validity of evidence, and authority, is enforced only by the guard. That is the precise content of calling the disposition calculus a discipline over the kernel.

3.5 Forward simulation and backward adequacy

Lemma 3.4 (Records occur once)**.** *If* replay(H) *is defined, no record occurs twice in* H*.*

Proof. A repeated record has the same subject as its first occurrence, so both lie in one projection. Every clause of δ other than discharge and timeout moves strictly along the acyclic lifecycle or out of $\bot$, so the second occurrence would meet a state from which its clause is undefined. A repeated discharge or timeout of the same obligation is rejected by the set of settled obligations carried beside the state. □

Theorem 3.5 (Forward simulation)**.** *If* replay(H) *is defined and* $G_\rho(H)$ *holds, then* $[\![H]\!]; [\![\rho]\!]_H$ *is a kernel-admissible history whose decoding is* $H \cdot \rho$*.*

Proof. By induction on H the prefix $[\![H]\!]$ is admissible and decodes to H. For the new events: ρ has not been popped, by Lemma 3.4 applied to $H \cdot \rho$, whose replay is defined because the guard includes $\rho \in \Omega^1_c(H)$. It has not been refused, because the only refusals of options with subject c are those emitted by a terminal step of c, and a terminal c has an empty one-step space, contradicting the guard. So $\mathsf{Pop}(\rho)$ is admissible. Each Bind couples ρ to a record already popped. Each Refuse is admissible unconditionally. Decoding lists the popped records in order, which is H followed by ρ. □

Corollary 3.6 (Operational forward simulation)**.** *If* C *is consistent and* $C \xrightarrow{\alpha} C'$ *appends* ρ*, then* $[\![H]\!] \xRightarrow{[\![\rho]\!]_H} [\![H']\!]$ *in the kernel.*

Proof. The rule that fired has the guard's premises: its lifecycle and data premises give $\rho \in \Omega^1_c(H)$ because replay of H reconstructs A (consistency), and its authority premise is Theorem 9.9. Apply Theorem 3.5. □

The converse direction is where the phrase "discipline over the kernel" earns its meaning. Without it, the translation might embed disposition steps into a more permissive system without the permissiveness showing.

Call a kernel extension β of $[\![H]\!]$ *typed-guarded* if it has the shape $[\![\rho]\!]_H$ for some record ρ and $G_\rho(H)$ holds. Let U be the *universal process* over a finite set of names and values: the replication of the sum of every prefix, each followed by $\mathbf{0}$. It can attempt every disposition step, and it performs exactly those whose premises hold.

Theorem 3.7 (Backward adequacy)**.** *Let $C = \langle U, S, A, O, H\rangle$ be consistent, and let $[\![H]\!]; \beta$ be kernel admissible with β typed-guarded, of shape $[\![\rho]\!]_H$. Then there is a transition $C \xrightarrow{\alpha} C'$ appending ρ, and $[\![H]\!]; \beta = [\![H']\!]$.*

Proof. The guard is the conjunction of the premises of the rule for ρ, read against replay of H, which by consistency is the runtime state. The universal process offers the corresponding prefix, which by the guard is enabled; firing it appends ρ. The equation is the definition of the translation. □

Together the two theorems say that, under the opaque reading and at the level of ledgers, translated disposition histories are exactly the kernel histories whose steps pass their guards. The disposition calculus adds no causal primitive and removes no kernel history that its guards admit. It is in this sense a *conservative guarded presentation* of the kernel.

Three limits should be kept in view. First, the theorems are relative to the universal process; a particular program offers fewer steps, as every program does, and that is not a failure of adequacy. Second, they hold under the opaque reading, in which evaluation and verification results are inputs checked by the guard. They do not show that the kernel performs those computations. Third, the guard lives in the metalanguage: the kernel does not compute it.

3.6 Opaque and transparent readings

Under the *opaque* reading, the results of evaluation and verification enter as recorded data whose correctness is part of the guard. That is the reading of the theorems above, and it supports every operational and ledger result of the book without any claim about what the kernel can compute.

Under the *transparent* reading, the value v of a transformation is itself produced inside the kernel by its encoding of application, $\mathsf{Collapse}_{\mathrm{subst}}(\mathsf{Bind}(f, b))$, and similarly for verifiers. Only this reading would support the claim that Spherepop itself performs the computation. It requires a theorem the book does not yet have: that the kernel's application encoding agrees with evaluation in the lambda layer, $\mathsf{Collapse}_{\mathrm{subst}}(\mathsf{Bind}(f, b)) = v$ exactly when $f\, b \Downarrow v$. It is stated as a conjecture and listed in Appendix E.

3.7 What the translation preserves

Proposition 3.8 (Compositionality)**.** $[\![H \cdot H']\!] = [\![H]\!]; [\![H']\!]_H$*, where the second translation is taken record by record at the successive prefixes.*

Proof. Induction on H' using the defining clause for histories. □

Proposition 3.9 (Locality)**.** *The translation of a record pops and binds only options with its own subject, except that a compensation record also binds to the last record of its predecessor.*

Proof. Inspection of Definition 3.3. □

From these the translation preserves commitment identity (distinct commitments occupy disjoint chains of options, and no clause identifies two of them); causal order (records of one commitment are chained by Bind in ledger order, while independent records touch disjoint options and may be interleaved, so the dependence order of the ledger is carried to the kernel and nothing is added); refusal records (each disposition refusal is a kernel refusal in an append-only history); and declared observations (each (c, k, o) in O corresponds to exactly one popped collapse record carrying k). Finally, the commitment map itself has a kernel reading: δ is an observation rule on histories, and $A = \mathsf{Collapse}_\delta([\![H]\!])$ after decoding. The disposition state is rendered from the history, not stored beside it.

3.8 Decisions fixed by this chapter

1. *No new causal primitive.* Under the opaque reading and at the level of ledgers, the disposition calculus is a conservative guarded presentation of the kernel (Theorems 3.5 and 3.7). Transformation, verification, and discharge are guarded Bind ∘ Pop. Whether the kernel also performs the computations, the transparent reading, is conjectural.

2. *Refusal is available from every live state* and translates to a choice that documents the whole remaining option space of c as inadmissible.

3. *Collapse is always indexed by a rule k*, recorded in the ledger, and means a committed observation, with observable label $\langle k, o\rangle$.

4. *Verification records both outcomes and disposes of neither.* Success appends a Verify record and leads to verified; failure appends a VerificationFailed record carrying an authenticated certificate and leads to the live state verificationFailed, from which only an authorized refusal exits.

5. *Plain pop requires no authority.* Introducing an identity commits no principal to anything; the first attributed act on a commitment is its binding, discharge, or refusal.

6. *Replay reconstructs the commitment map and observation set*, not the store. The store is reconstructible exactly when every state-affecting input is recorded.

7. *Discharge and timeout are kernel-derived.* Each is a guarded Bind ∘ Pop of authenticated evidence onto a waiting commitment's history, not a fifth primitive and not an annotation. Neither ends a commitment.

Chapter 4

Encodings and Their Criteria

A comparison between calculi is only as good as the translation it rests on. Informal analogies between operators are easy to produce and hard to refute, and the literature on process calculi has developed a discipline for doing better: state a translation, state the criteria it satisfies, and state what it fails to preserve. This chapter fixes the criteria used throughout the book and the three outcomes a comparison can have.

4.1 Calculi and translations

For the purposes of comparison, a *calculus* is a set of terms with a labelled transition relation $\xrightarrow{\mu}$, a set of names occurring in terms, a distinguished internal label τ, and a *success* predicate $S \Downarrow$, meaning that S can reach a term exhibiting a designated success observable. Write $\Rightarrow$ for the reflexive and transitive closure of internal steps, and $\stackrel{\mu}{\Rightarrow}$ for $\Rightarrow\xrightarrow{\mu}\Rightarrow$.

A *translation* $[\![\cdot]\!]$ from a source calculus to a target calculus comes with a *renaming policy* φ that assigns to each source name a tuple of target names, and, when the comparison is labelled, an *action translation* $\widehat{\mu}$ taking each source label to a target label or to τ.

Gorla's framework is stated for reduction semantics. The book compares calculi that are most naturally presented as labelled transition systems, so it uses the labelled forms of his criteria below. Where the difference matters, the chapter concerned says so.

4.2 The criteria

Compositionality.
For each source operator op of arity k and each finite set of names N there is a target context C^N_{op} with k holes such that $[\![\mathrm{op}(S_1,\dots,S_k)]\!] = C^N_{\mathrm{op}}([\![S_1]\!],\dots,[\![S_k]\!])$ whenever the free names of the S_i are N. A translation is *homomorphic* on an operator when the context is that operator itself, a strictly stronger requirement.

Name invariance.
For every injective renaming σ of source names there is a target renaming σ' with $\varphi \circ \sigma = \sigma' \circ \varphi$ and $[\![\sigma S]\!] = \sigma' [\![S]\!]$.

Operational correspondence.
Completeness: if $S \stackrel{\mu}{\Rightarrow} S'$ then $[\![S]\!] \stackrel{\widehat{\mu}}{\Rightarrow} \asymp [\![S']\!]$. *Soundness*: if $[\![S]\!] \stackrel{\nu}{\Rightarrow} T$ then there are S'

and μ with $\widehat{\mu} = \nu$, $S \stackrel{\mu}{\Longrightarrow} S'$, and $T \Rightarrow \asymp [\![S']\!]$. Here $\asymp$ is a fixed equivalence on target terms, used to absorb administrative residue.

Divergence reflection.
If $[\![S]\!]$ has an infinite sequence of internal steps, so has S.

Success sensitiveness.
$S \Downarrow$ if and only if $[\![S]\!] \Downarrow$.

Compositionality and name invariance are structural; the other three concern behaviour. A translation that satisfies none of the structural criteria can relate any two Turing-complete calculi, by compiling one into an interpreter for the other, which is why the structural criteria are what give a comparison its content.

Full abstraction.
For chosen equivalences $\approx_s$ on source terms and $\approx_t$ on target terms, $S_1 \approx_s S_2$ if and only if $[\![S_1]\!] \approx_t [\![S_2]\!]$.

Full abstraction is not one of Gorla's criteria, and it is not a consequence of them. It depends entirely on the two equivalences chosen, and a translation may be fully abstract for one pair and not for another. Every claim of full abstraction in the book names its equivalences.

4.3 The criteria are independent

Each criterion rules out translations that the others admit, and it is worth seeing this concretely, since a comparison that states some criteria and not others is claiming exactly the ones it states. The examples are translations of CCS into itself, with success a designated action $\checkmark$ reached by internal steps.

Complete but not sound.
Translate choice as parallel composition, $[\![S + T]\!] = [\![S]\!] \mid [\![T]\!]$, and every other operator homomorphically. The translation is compositional and name invariant, and it is operationally complete: whatever $S + T$ does, one side of the parallel composition can do. It is not sound. The translation of $a.\mathbf{0} + b.\mathbf{0}$ can perform a and then b, which the source cannot, because the source discards b when it performs a. Completeness alone admits a translation that adds behaviour.

Success sensitive but not compositional.
Let $[\![S]\!] = \checkmark$ if $S \Downarrow$ and $\mathbf{0}$ otherwise. It is success sensitive by construction. It is not compositional, and the proof is short. Suppose $[\![S \mid T]\!] = C([\![S]\!], [\![T]\!])$ for a fixed context C at the name set $\{a\}$. Then $a.\checkmark \mid \bar{a}.\mathbf{0}$ succeeds and $a.\checkmark \mid \mathbf{0}$ does not, while $[\![a.\checkmark]\!] = [\![\bar{a}.\mathbf{0}]\!] = [\![\mathbf{0}]\!] = \mathbf{0}$, so $C(\mathbf{0}, \mathbf{0})$ would have to both succeed and fail. This is the interpreter of the structural criteria's motivation reduced to one bit: success sensitiveness alone can be met by computing the answer and outputting it.

Compositional but not success sensitive.
Let $[\![S]\!] = \mathbf{0}$ for every S, the context for each operator discarding its holes. It is compositional, name invariant, divergence reflecting, and vacuously sound, since the target never moves. It is neither complete nor success sensitive. Structure alone is cheap.

Operationally corresponding but not divergence reflecting.
Translate each internal prefix through a retry loop, $[\![\tau.P]\!] = (\nu r)\big(\bar{r} \mid !\, r.(\tau.\bar{r} + \tau.[\![P]\!])\big)$, and every other operator homomorphically. Up to weak equivalence the translation is complete and sound, and it preserves success, but $[\![\tau.\mathbf{0}]\!]$ has an infinite internal run that $\tau.\mathbf{0}$ does not. This is the failure that an untyped encoding of the disposition calculus into the π-calculus would suffer (Chapter 12), and divergence reflection is the criterion that detects it.

So none of the criteria follows from the others, and every theorem in the book that asserts an encoding names the criteria it meets. When it asserts a separation, it names the criteria whose conjunction is impossible; Theorem 10.5, for example, uses only compositionality and success sensitiveness, and the second example above shows why both are needed.

4.4 Encodings and erasures

The book translates in both directions, and the two directions answer different questions.

An *encoding* takes a classical calculus into the disposition calculus. A good encoding shows that the disposition calculus can express what the classical calculus expresses, and the criteria it fails show where it cannot.

An *erasure* takes the disposition calculus to a classical calculus by forgetting structure: annotations, authority, evidence, the ledger. An erasure with good forward correspondence shows that the classical calculus captures the *shape* of disposition behaviour; the pairs of configurations that it identifies but the disposition calculus distinguishes show what the classical calculus forgets. Erasures are expected to fail full abstraction, and the interesting output is the smallest pair that shows it.

4.5 Three outcomes

Every comparison in Parts III and VI ends in one of three outcomes, and states which.

- *Equivalence*: translations in both directions satisfying the stated criteria.
- *Strict inclusion*: a translation in one direction satisfying the criteria, and a *separation result* in the other, a proof that no translation satisfying stated criteria exists, in the manner of Palamidessi's separation of the synchronous and asynchronous π-calculi.
- *Incomparability*: separation results in both directions, or a demonstration that the two calculi answer different questions about different objects, in which case the book says so rather than forcing a verdict.

A separation result is a statement that no translation meeting the stated criteria can exist, not a report that none has been found. Where the book has only the second, it says so and lists the question in Appendix E.

4.6 Why Turing completeness says little

The kernel inherits Turing completeness from its encoding of the lambda calculus, and the disposition calculus inherits it from its lambda layer through fixed points. Neither fact bears

on any comparison in this book. Turing completeness is a statement about the functions a calculus can compute, and it is preserved by translations that are neither compositional nor name invariant. The comparisons here are about structure: whether synchronization, refusal, fresh names, causality, and observation are expressed by corresponding constructs, and what a translation must add or forget to make them correspond. Two calculi can both be Turing complete and be separated by every one of those questions.

Part II

Terms and Names

Chapter 5

Syntax and Binding

5.1 Two syntactic categories

The calculus separates expressions from processes. Expressions compute; processes interact and alter disposition. The expression grammar is

$$\begin{aligned} e ::= {} & x \mid \lambda x{:}A.e \mid e\,e \mid \langle e, e\rangle \mid \pi_1 e \mid \pi_2 e \\ & \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \cdots . \end{aligned}$$

The process grammar is

$$\begin{aligned} P, Q ::= {} & \mathbf{0} \mid \mathsf{pop}\ q\ \mathsf{as}\ c.P \mid \mathsf{pop}_a\ q\ \mathsf{as}\ c\ \mathsf{after}\ d.P \\ & \mid \mathsf{refuse}_a\ c\ \mathsf{because}\ r.P \mid \mathsf{bind}_a\ c\ \mathsf{to}\ b.P \mid \mathsf{transform}\ c\ \mathsf{with}\ f.P \\ & \mid \mathsf{verify}_a\ c\ \mathsf{using}\ g\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid \mathsf{collapse}_{a,k}\ c.P \\ & \mid \mathsf{await}_a\ c\ \omega.P \mid \mathsf{await}_a\ c\ \omega.P\ \mathsf{within}_b\ t\ \mathsf{else}\ Q \\ & \mid P \mid Q \mid (\nu c)P \mid P + Q \mid !P. \end{aligned}$$

Subscripts name the acting principal a and, for collapse, the observation rule k. Pop and transformation are unattributed: the first only introduces an identity, and the second is a lambda-layer computation. The compensating pop $\mathsf{pop}_a\ q\ \mathsf{as}\ c\ \mathsf{after}\ d$ introduces a successor c to a terminal commitment d and is attributed, since reconsidering a disposition is itself an act someone must be entitled to perform.

Verification carries two continuations because it may decide either way. A single-continuation form would leave a negative decision without a successor state, and the commitment would sit in transformed with nothing recorded, which is exactly the unrecorded dead end the lifecycle was designed to exclude.

Collapse names no witness. It uses the witness recorded in the commitment's verified state, so a program cannot cite the wrong one, and a well-typed program cannot stall on a mismatch between the witness it holds and the one the state holds. The prefix $\mathsf{await}_a\ c\ \omega.P$ suspends the process until principal a discharges an external obligation ω on which commitment c depends, such as a committee decision or a response from an outside service. It is the calculus's positive form of waiting: a process that is waiting says so, names the commitment, and names who it is waiting for, rather than simply failing to move. Waiting itself writes nothing to the ledger; only a successful discharge does. The deadline form $\mathsf{await}_a\ c\ \omega.P\ \mathsf{within}_b\ t\ \mathsf{else}\ Q$ adds an explicit

branch: either a discharges ω and the process continues as P, or a clock authority b certifies that the interval t passed without discharge and the process continues as Q. A timeout does not convert silence into refusal. It records that the obligation went unanswered, and it is up to an authorized rule in Q to decide whether the commitment then waits further, is compensated, or is refused.

Replication creates new commitments on every unfolding. Because pop binds its identity, the law $!P \equiv P \mid !P$ together with α-conversion yields a fresh copy of each generative binder in each copy of P, and the pop rule's freshness premise forces distinct identities when those copies fire. No two instances ever share a commitment.

A negative verification does not refuse the commitment. The failure branch records the verifier's certificate and continues as Q with the commitment live in verificationFailed; if the commitment is to end, Q must contain a refusal by a principal entitled to refuse. A verifier therefore changes the evidentiary state of a commitment without ever disposing of it.

The binder in $\mathsf{pop}\ q\ \mathsf{as}\ c.P$ scopes over P, as does the binder of the compensating pop and restriction $(\nu c)P$. Bound variables in lambda abstractions scope over their bodies. Commitment identities and expression variables form disjoint sorts, and they remain sorted even if a concrete implementation represents them with the same strings.

Definition 5.1 (Alpha-equivalence)**.** *Alpha-equivalence, written $M \equiv_\alpha N$, is the least congruence identifying terms which differ only by a consistent capture-avoiding renaming of bound variables or bound commitment identities.*

5.2 Free names and substitution

The free commitment names of representative processes are defined by

$$\begin{aligned}
\mathrm{fn}(\mathbf{0}) &= \varnothing,\\
\mathrm{fn}(P \mid Q) &= \mathrm{fn}(P) \cup \mathrm{fn}(Q),\\
\mathrm{fn}((\nu c)P) &= \mathrm{fn}(P) \setminus \{c\},\\
\mathrm{fn}(\mathsf{pop}\ q\ \mathsf{as}\ c.P) &= \mathrm{fn}(q) \cup (\mathrm{fn}(P) \setminus \{c\}),\\
\mathrm{fn}(\mathsf{pop}_a\ q\ \mathsf{as}\ c\ \mathsf{after}\ d.P) &= \{d\} \cup \mathrm{fn}(q) \cup (\mathrm{fn}(P) \setminus \{c\}),\\
\mathrm{fn}(\mathsf{bind}_a\ c\ \mathsf{to}\ b.P) &= \{c\} \cup \mathrm{fn}(b) \cup \mathrm{fn}(P),\\
\mathrm{fn}(\mathsf{verify}_a\ c\ \mathsf{using}\ g\ \mathsf{then}\ P\ \mathsf{else}\ Q) &= \{c\} \cup \mathrm{fn}(g) \cup \mathrm{fn}(P) \cup \mathrm{fn}(Q),\\
\mathrm{fn}(\mathsf{collapse}_{a,k}\ c.P) &= \{c\} \cup \mathrm{fn}(P), \qquad \mathrm{fn}(\mathsf{await}_a\ c\ \omega.P) = \{c,\omega\} \cup \mathrm{fn}(P),\\
\mathrm{fn}(\mathsf{await}_a\ c\ \omega.P\ \mathsf{within}_b\ t\ \mathsf{else}\ Q) &= \{c,\omega\} \cup \mathrm{fn}(P) \cup \mathrm{fn}(Q).
\end{aligned}$$

Expression substitution $e[v/x]$ and name substitution $P[d/c]$ are capture avoiding. Whenever a substitution would capture a free name, a bound name is first replaced by a fresh one.

Lemma 5.2 (Freshening)**.** *For every finite set of names N and term M, there exists $M' \equiv_\alpha M$ whose bound names are disjoint from N.*

Proof. The syntax tree of M contains finitely many binders. Rename them one at a time using names outside the union of N, the free names of M, and the names already selected. The supply of names is countably infinite, while the excluded set at each step is finite. □

5.3 Structural congruence

Structural congruence identifies processes whose difference is purely administrative:

$$P \mid \mathbf{0} \equiv P, \qquad P \mid Q \equiv Q \mid P, \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R),$$
$$P + Q \equiv Q + P, \qquad !P \equiv P \mid !P, \qquad (\nu c)(\nu d)P \equiv (\nu d)(\nu c)P,$$
$$(\nu c)\mathbf{0} \equiv \mathbf{0}, \qquad (\nu c)(P \mid Q) \equiv P \mid (\nu c)Q \quad \text{if } c \notin \mathrm{fn}(P),$$

together with α-equivalence. No structural law may delete a recorded event or identify distinct commitment identities. Structural congruence rearranges a process before an event occurs; it does not rewrite the history afterward.

Theorem 5.3 (Equivariance)**.** *For every capture-avoiding permutation of names ρ,*

$$C \xrightarrow{\alpha} C' \quad \Longrightarrow \quad \rho(C) \xrightarrow{\rho(\alpha)} \rho(C').$$

Proof. By induction on the transition derivation of Chapter 9. Each primitive rule compares names only for equality, freshness, or membership in a finite support. Permutations preserve all three properties. The contextual cases are immediate from the induction hypothesis, and the congruence case follows from closure of structural congruence under capture-avoiding renaming. □

The theorem is stated for permutations rather than arbitrary renamings because freshness is preserved by bijections and not in general by many-to-one maps. It licenses every later proof to choose fresh names without the result depending on a particular identifier generator.

Chapter 6

Concrete Syntax and Unique Parsing

Semantic disagreements must not hide inside parsing disagreements. If one source text could be read as two processes, a theorem about processes would say nothing definite about programs. This chapter fixes a concrete syntax and proves that it is read one way only. The claim is made at three levels of strength, and they are kept apart: lexical analysis is deterministic; every accepted token sequence has exactly one derivation; and a bounded check confirms that the implementation and printer agree with that theory. The third does not stand in for the second.

6.1 Lexical classes

Tokens fall into disjoint classes, and each class is recognizable from its first character:

Class	Form	Example
commitment names (CN)	`#`NAME	`#c`
principals (PR)	`@`NAME	`@a`
obligations (OB)	`?`NAME	`?w`
observation rules (RL)	`%`NAME	`%alloc`
intervals (TM)	`~`NAME	`~t`
reasons (RS)	quoted, with backslash escapes	`"late"`
values, functions, requests (VAL)	lowercase-initial NAME, not a keyword	`f`
keywords and punctuation	fixed strings	`verify`, `|`

Here NAME is an ASCII letter or underscore followed by ASCII letters, digits, and underscores. The classes are chosen to make the sorts of the abstract syntax visible in the text: a commitment name cannot stand where a principal is expected, because the two cannot even be the same token.

Lemma 6.1 (Unique tokenization)**.** *Every string has at most one tokenization.*

Proof. After whitespace, the first character determines the class: the five sigils, the quotation mark, a lowercase letter, or a punctuation character, and these are pairwise distinct. Within a class the token is the longest match, which is unique. A lowercase identifier is a keyword

exactly when it appears in the fixed keyword list. A quoted reason ends at the first unescaped quotation mark. □

Failure certificates, witnesses, and ledger records have no concrete syntax at all. They are runtime objects, produced by verifiers and appended by rules, and no production of the grammar below mentions them. That is a fact about the grammar's formation, not the outcome of a test: no source program can contain a certificate, so none can forge one.

6.2 The grammar

The grammar is stratified by precedence, with choice loosest, parallel composition tighter, and prefixes, restriction, and replication tightest:

$$P ::= S\ P_1 \quad P_1 ::= \texttt{+}\ P \mid \varepsilon$$
$$S ::= U\ S_1 \quad S_1 ::= \texttt{|}\ S \mid \varepsilon$$

$$\begin{aligned} U ::= {} & \texttt{0} \mid \texttt{(}\ P\ \texttt{)} \mid \texttt{nu}\ CN\ \texttt{.}\ U \mid \texttt{!}\ U \mid \texttt{pop}\ POP \\ & \mid \texttt{bind}\ \texttt{[}PR\texttt{]}\ CN\ \texttt{to}\ VAL\ \texttt{.}\ U \mid \texttt{transform}\ CN\ \texttt{with}\ VAL\ \texttt{.}\ U \\ & \mid \texttt{refuse}\ \texttt{[}PR\texttt{]}\ CN\ \texttt{because}\ RS\ \texttt{.}\ U \mid \texttt{collapse}\ \texttt{[}PR\texttt{,}\ RL\texttt{]}\ CN\ \texttt{.}\ U \\ & \mid \texttt{verify}\ \texttt{[}PR\texttt{]}\ CN\ \texttt{using}\ VAL\ \texttt{then}\ U\ \texttt{else}\ U \mid \texttt{await}\ \texttt{[}PR\texttt{]}\ CN\ OB\ AW \\ POP ::= {} & VAL\ \texttt{as}\ CN\ \texttt{.}\ U \mid \texttt{[}PR\texttt{]}\ VAL\ \texttt{as}\ CN\ \texttt{after}\ CN\ \texttt{.}\ U \\ AW ::= {} & \texttt{.}\ U \mid \texttt{within}\ \texttt{[}PR\texttt{]}\ TM\ \texttt{then}\ U\ \texttt{else}\ U \end{aligned}$$

Choice and parallel composition associate to the right, by the direction of their recursion. Parentheses reset precedence.

One scoping decision needs stating because readers from other traditions may expect otherwise. The continuation after a prefix's dot, and each branch after `then` and `else`, is a unary process U, not an arbitrary process. So

$$\texttt{pop q as \#c . 0 | 0} \quad \text{reads as} \quad (\texttt{pop}\ q\ \texttt{as}\ c.\mathbf{0}) \mid \mathbf{0},$$

and a parallel continuation must be parenthesized, `pop q as #c . (0 | 0)`. The same holds for the branches of verification and of a deadline. This is the usual convention of CCS, where prefix binds more tightly than parallel composition, and it is what makes the two-branch forms unambiguous: every `then` has exactly one `else`, and no `else` can be claimed by more than one `then`. The deadline form is distinguished from the plain `await` by its second token after the obligation, `within` rather than the dot, so there is no dangling-else problem there either.

6.3 Unique derivations

For a nonterminal X and one of its alternatives β, the *predict set* is FIRST(β), together with FOLLOW(X) if β can derive the empty string. A grammar is LL(1) when, for each nonterminal, the predict sets of its alternatives are pairwise disjoint. For the grammar above:

- P_1: the alternative `+`P is predicted by `+`; the empty alternative by FOLLOW(P_1) = FOLLOW(P) = {`)`, \$}.

- S_1: `|` S is predicted by `|`; the empty alternative by FOLLOW$(S) = \{$`+`, `)`, `$`$\}$.
- U: each of the eleven alternatives begins with a distinct terminal.
- *POP*: *VAL* versus `[`.
- *AW*: `.` versus `within`.

The laboratory computes the same sets mechanically from the grammar and finds no conflict (property P-ParserLL1).

Theorem 6.2 (Unique parsing)**.** *Every token sequence in the language of the grammar has exactly one derivation. Consequently, by Lemma 6.1, every accepted source text denotes exactly one abstract syntax tree.*

Proof. Suppose a token sequence has two distinct leftmost derivations. Consider the first step at which they differ. Both expand the same leftmost nonterminal X, with different alternatives $\beta \neq \beta'$, and the remaining input begins with the same token t. In each derivation, t is either the first token derived from the chosen alternative, or, if the alternative derives the empty string there, a token that follows X. So t lies in the predict sets of both β and β', contradicting the LL(1) property. The conversion from derivation to abstract syntax tree is a function, so the tree is unique too. □

Structural congruence plays no part in this theorem. The texts `0 | pop q as #c . 0` and `pop q as #c . 0 | 0` have different, unique trees; that those trees denote congruent processes is a fact about the semantics, not an ambiguity of the syntax.

6.4 Adequacy of the printer

Let show(P) print a tree with the fewest parentheses its precedence requires, and let Concrete(P) be the set of texts obtained from show(P) by wrapping any subterms in additional, redundant parentheses.

Theorem 6.3 (Adequacy)**.** parse$(s) = P$ *if and only if* $s \in$ Concrete(P).

Proof. From right to left, by induction on P: the minimal printing parses to P because it inserts parentheses exactly where the precedence stratification requires them, and a redundant pair around a subterm is the production $U \to$ `(` P `)`, whose tree is the tree of the enclosed process. From left to right, by induction on the unique derivation of s: the conversion to a tree discards only parentheses and the empty expansions of P_1 and S_1, so s differs from show(parse(s)) only by parentheses around subterms, which is membership in Concrete(parse(s)). □

6.5 The executable check

The reference parser is generated from the same grammar: a table-driven predictive parser whose table is the one computed for the LL(1) check, followed by a conversion from parse trees to abstract syntax trees. There is one grammar in the implementation, not two, so the grammar that is analyzed is the grammar that is parsed.

Beyond the general property, the module checks bounded properties over every abstract syntax tree with at most five constructors, 9,469 distinct trees over a small alphabet of atoms. The alphabet can be small because parsing depends only on token classes, not on which name or value appears. For every such P:

- P-ParserUnique: show(P) has exactly one derivation, counted by a chart that enumerates all derivations rather than following the table;
- P-ParserRoundTrip: parse(show(P)) = P, and parse of a fully redundantly parenthesized printing of P is also P;
- P-ParserNormalize: show(parse(show(P))) = show(P), and parse(show(parse(s))) = parse(s) for the redundantly parenthesized s, so redundant parentheses disappear without changing the tree.

Eleven named cases pin down the decisions of this chapter with expected trees, among them prefix scope, the relative precedence of choice and parallel composition, right associativity, the unary branches of verification, the deadline form, and escaped reasons (P-ParserNamed). Seventeen malformed inputs are rejected, among them an unquoted value in the reason position, an unterminated reason, a keyword in a value position, an empty parenthesized process, an extra `else`, a malformed rule index, an obligation in a commitment position, and a deadline lacking its clock principal (P-ParserRejects). As a negative control, adding an ambiguous alternative to the grammar produces both an LL(1) conflict and a sentence with two derivations; the checks detect the ambiguity they are meant to detect.

These checks establish that the implementation and printer agree with the theory over the enumerated class. The universal statement is Theorem 6.2, which rests on the LL(1) property and not on the enumeration.

Chapter 7

The Lambda Layer

Transformations and verifiers are lambda terms, and the operational rules consume their evaluation as a premise, $f\ b \Downarrow v$. This chapter fixes that evaluation relation, proves the two properties of it the rest of the book depends on, determinism and the dichotomy between evaluation and divergence for well-typed terms, and asks what a history retains that a normal form does not.

7.1 Terms and types

Transformations and verifiers are closed terms of the unrestricted fragment of the expression language of Chapter 5, extended with constants, primitive operations, and a fixed-point operator. Types and terms are

$$\begin{aligned} A, B &::= \iota \mid A \to B \mid A \times B \mid A + B, \\ e &::= x \mid \kappa \mid p\ e \mid \lambda x{:}A.e \mid e\ e \mid \langle e, e \rangle \mid \pi_1 e \mid \pi_2 e \\ &\quad \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x \Rightarrow e \mid \mathsf{inr}\ y \Rightarrow e \mid \mathsf{fix}\, x{:}A.e, \end{aligned}$$

where ι ranges over base types (natural numbers, strings, the unit type $\mathbf{1}$ with its one constant $*$), κ over constants of base type, and p over primitive operations, each with a fixed type $\iota_1 \to \iota_2$ or $\iota_1 \times \iota_2 \to B$ and a total function $\hat{p}$ on constants of its argument type. Booleans are $\mathbf{1} + \mathbf{1}$, with $\mathsf{inl}\, *$ for true. The typing rules are the standard ones:

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\kappa \text{ of type } \iota}{\Gamma \vdash \kappa : \iota} \qquad \frac{p : A \to B \quad \Gamma \vdash e : A}{\Gamma \vdash p\ e : B}$$

$$\frac{\Gamma, x{:}A \vdash e : B}{\Gamma \vdash \lambda x{:}A.e : A \to B} \qquad \frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B}$$

$$\frac{\Gamma \vdash e_1 : A_1 \quad \Gamma \vdash e_2 : A_2}{\Gamma \vdash \langle e_1, e_2 \rangle : A_1 \times A_2} \qquad \frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \pi_i\, e : A_i}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathsf{inl}\ e : A + B} \qquad \frac{\Gamma \vdash e : B}{\Gamma \vdash \mathsf{inr}\ e : A + B}$$

$$\frac{\Gamma \vdash e : A + B \quad \Gamma, x{:}A \vdash e_1 : C \quad \Gamma, y{:}B \vdash e_2 : C}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x \Rightarrow e_1 \mid \mathsf{inr}\ y \Rightarrow e_2 : C}$$

$$\frac{\Gamma, x{:}A \to B \vdash e : A \to B}{\Gamma \vdash \mathsf{fix}\, x{:}A \to B.e : A \to B}$$

The fixed-point operator is restricted to function types, the usual choice for a call-by-value language: it makes fix a way of defining recursive functions rather than a way of writing a divergent value of any type. A transformation from A to B is a closed term of type $A \to B$; a verifier's decision procedure is a closed term returning a boolean, with the witness or certificate it produces computed alongside.

7.2 Why big-step

The operational rules of the process layer do not watch a computation; they consume its result. The premise of the transformation rule is a judgement $f\, b \Downarrow v$, and the record it appends, $\mathsf{Transform}(c, f, b, v)$, names the function, the input, and the output, and nothing in between. A big-step relation is the relation of exactly that shape. A small-step presentation would define intermediate terms that the process layer never sees and the ledger never holds, and it would then need a second definition, the reflexive-transitive closure to a value, to state the premise the process rules actually use. Big-step semantics states that premise directly.

The choice has a price, and it is paid below. A big-step relation says what happens when evaluation succeeds and is silent otherwise. A term with no derivation might be one whose evaluation runs forever or one whose evaluation reaches a configuration no rule covers, and the relation alone does not distinguish them. The distinction matters here, because the two failures mean different things for a commitment: divergence leaves a $\mathsf{transform}$ prefix permanently disabled, which is a liveness failure the book assigns to Hypothesis 15.8; getting stuck would be a type error, which the book rules out. The lemma at the end of this chapter shows that for closed well-typed terms only the first can occur.

The choice also fixes what the opaque reading of Chapter 3 means. Under that reading a transformation is atomic: its derivation is evidence the implementation may keep, but not a history the calculus records. The transparent reading would record the derivation, and a small-step presentation is the natural one for that purpose.

7.3 Evaluation

Values are

$$u ::= \kappa \mid \lambda x{:}A.e \mid \langle u, u \rangle \mid \mathsf{inl}\ u \mid \mathsf{inr}\ u.$$

Evaluation is call by value, left to right, and its rules are

$$\textsc{E-Val}\ \frac{}{u \Downarrow u} \qquad \textsc{E-Prim}\ \frac{e \Downarrow \kappa}{p\ e \Downarrow \hat{p}(\kappa)} \qquad \textsc{E-PrimPair}\ \frac{e \Downarrow \langle \kappa_1, \kappa_2 \rangle}{p\ e \Downarrow \hat{p}(\kappa_1, \kappa_2)}$$

$$\textsc{E-App}\ \frac{e_1 \Downarrow \lambda x{:}A.e \quad e_2 \Downarrow u_2 \quad e[u_2/x] \Downarrow u}{e_1\ e_2 \Downarrow u} \qquad \textsc{E-Pair}\ \frac{e_1 \Downarrow u_1 \quad e_2 \Downarrow u_2}{\langle e_1, e_2 \rangle \Downarrow \langle u_1, u_2 \rangle} \qquad \textsc{E-Proj}_i\ \frac{e \Downarrow \langle u_1, u_2 \rangle}{\pi_i\ e \Downarrow u_i}$$

$$\textsc{E-Inl}\ \frac{e \Downarrow u}{\mathsf{inl}\ e \Downarrow \mathsf{inl}\ u} \qquad \textsc{E-Inr}\ \frac{e \Downarrow u}{\mathsf{inr}\ e \Downarrow \mathsf{inr}\ u} \qquad \textsc{E-Fix}\ \frac{}{\mathsf{fix}\, x{:}T.e \Downarrow \lambda z{:}A.\, (e[\mathsf{fix}\, x{:}T.e/x])\ z}$$

$$\textsc{E-CaseL}\ \frac{e \Downarrow \mathsf{inl}\ u \quad e_1[u/x] \Downarrow u'}{\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow e_1 \mid \mathsf{inr}\, y \Rightarrow e_2\ \Downarrow u'} \qquad \textsc{E-CaseR}\ \frac{e \Downarrow \mathsf{inr}\ u \quad e_2[u/y] \Downarrow u'}{\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow e_1 \mid \mathsf{inr}\, y \Rightarrow e_2\ \Downarrow u'}$$

where in E-Fix the type T is $A \to B$ and z is fresh. The fixed point unfolds to an abstraction, which is a value, so a recursive function is unfolded once per call rather than eagerly; that is what restricting fix to function types buys.

7.4 A complete derivation

Let $lt : \mathbb{N} \times \mathbb{N} \to \mathbf{1} + \mathbf{1}$ be the primitive comparison, and let

$$f = \lambda x{:}\mathbb{N} \times \mathbb{N}.\ \mathsf{case}\ lt\ x\ \mathsf{of}\ \mathsf{inl}\, y \Rightarrow \pi_2\, x \mid \mathsf{inr}\, z \Rightarrow \pi_1\, x,$$

the transformation that returns the larger component of a pair. It has type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$: the scrutinee has type $\mathbf{1} + \mathbf{1}$ by the primitive rule, both branches have type $\mathbb{N}$ by projection from x, and the abstraction rule closes the derivation. Its evaluation on $b = \langle 3, 5\rangle$ is the following derivation, read from the leaves down; each line cites its rule and the lines it uses as premises.

(1)	$f \Downarrow f$	E-Val
(2)	$\langle 3,5\rangle \Downarrow \langle 3,5\rangle$	E-Val
(3)	$lt\ \langle 3,5\rangle \Downarrow \mathsf{inl}\, *$	E-PrimPair (2), $\widehat{lt}(3,5) = \mathsf{inl}\, *$
(4)	$\pi_2\ \langle 3,5\rangle \Downarrow 5$	E-Proj$_2$ (2)
(5)	$\mathsf{case}\ lt\ \langle 3,5\rangle\ \mathsf{of}\ \mathsf{inl}\, y \Rightarrow \pi_2\ \langle 3,5\rangle \mid \mathsf{inr}\, z \Rightarrow \pi_1\ \langle 3,5\rangle \ \Downarrow 5$	E-CaseL (3), (4)
(6)	$f\ \langle 3,5\rangle \Downarrow 5$	E-App (1), (2), (5)

In line (4) the substitution $(\pi_2\ \langle 3,5\rangle)[*/y]$ is the identity, since y does not occur in the branch. Line (5) is the body of f with $\langle 3,5\rangle$ substituted for x, which is the third premise E-App requires. The process step that consumes this derivation is

$$\mathsf{transform}\ c\ \mathsf{with}\ f, \qquad \mathsf{bound}(q, \langle 3,5\rangle) \longrightarrow \mathsf{transformed}(\langle 3,5\rangle, 5),$$

and the record it appends is $\mathsf{Transform}(c, f, \langle 3,5\rangle, 5)$. The six lines above are what the opaque reading discards and the transparent reading would keep.

7.5 Evaluation, divergence, and being stuck

There are three ways for a closed term e to relate to evaluation. It may *evaluate*, $e \Downarrow u$ for some u. It may *diverge*, written $e \Uparrow$, defined coinductively by the rules

$$\frac{e_1 \Uparrow}{e_1\, e_2 \Uparrow} \qquad \frac{e_1 \Downarrow u_1 \quad e_2 \Uparrow}{e_1\, e_2 \Uparrow} \qquad \frac{e_1 \Downarrow \lambda x{:}A.e \quad e_2 \Downarrow u_2 \quad e[u_2/x] \Uparrow}{e_1\, e_2 \Uparrow} \qquad \frac{e_1 \Uparrow}{\langle e_1, e_2\rangle \Uparrow} \qquad \frac{e_1 \Downarrow u_1 \quad e_2 \Uparrow}{\langle e_1, e_2\rangle \Uparrow}$$

$$\frac{e \Uparrow}{\pi_i\, e \Uparrow} \qquad \frac{e \Uparrow}{p\, e \Uparrow} \qquad \frac{e \Uparrow}{\mathsf{inl}\, e \Uparrow} \qquad \frac{e \Uparrow}{\mathsf{inr}\, e \Uparrow} \qquad \frac{e \Uparrow}{\mathsf{case}\ e\ \mathsf{of} \cdots \Uparrow}$$

$$\frac{e \Downarrow \mathsf{inl}\, u \quad e_1[u/x] \Uparrow}{\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow e_1 \mid \cdots \Uparrow} \qquad \frac{e \Downarrow \mathsf{inr}\, u \quad e_2[u/y] \Uparrow}{\mathsf{case}\ e\ \mathsf{of} \cdots \mid \mathsf{inr}\, y \Rightarrow e_2 \Uparrow}$$

read as the greatest relation closed under them, so that an infinite derivation is allowed. The recursive function $\Omega = \mathsf{fix}\, g{:}\mathbb{N} \to \mathbb{N}.\, \lambda n{:}\mathbb{N}.\, g\, n$ gives $\Omega\, 0 \Uparrow$: the third application rule applies with the body again an application of the unfolded Ω to 0, and the derivation repeats forever. Finally, e may be *stuck*: neither evaluating nor diverging. The term $\pi_1\, (\lambda x{:}\iota.x)$ is stuck, since its subterm evaluates to an abstraction and no projection rule applies to one; so is $\mathsf{case}\ \langle 3,5\rangle\ \mathsf{of} \cdots$. Both are ill typed, and that is not a coincidence.

Lemma 7.1 (Evaluation or divergence)**.** *Let e be closed with $\vdash e : A$. Then exactly one of the following holds: $e \Downarrow u$ for a unique value u, and then $\vdash u : A$; or $e \Uparrow$. In particular no closed well-typed term is stuck.*

Proof. At most one. Uniqueness of u is Lemma 7.2 below. That $e \Downarrow u$ and $e \Uparrow$ exclude each other follows by induction on the derivation of $e \Downarrow u$: in each rule, every divergence rule for the same term form requires some premise to diverge that the evaluation derivation shows to evaluate, or, when the rules select different branches, a scrutinee to evaluate to a different injection, which determinism forbids.

At least one. Let D be the set of closed well-typed terms that have no evaluation derivation. It suffices to show that every $e \in D$ is the conclusion of some divergence rule whose divergent premise is again in D and whose evaluation premises hold; then D is closed under the rules read backwards, and by coinduction every member of D diverges. Values evaluate, so e is not a value, and it is not a variable, being closed. Suppose $e = e_1\, e_2$. If $e_1 \in D$ the first rule applies. Otherwise $e_1 \Downarrow u_1$, and by preservation, proved as in Chapter 14 from the substitution lemmas, $\vdash u_1 : A \to B$, so by canonical forms $u_1 = \lambda x{:}A.e'$. If $e_2 \in D$ the second rule applies. Otherwise $e_2 \Downarrow u_2$ with $\vdash u_2 : A$, and $e'[u_2/x]$ is closed and well typed by substitution; it has no evaluation derivation, or E-App would give one for e, so it is in D and the third rule applies. The pair, projection, injection, and primitive cases are the same argument with fewer premises; for projections and primitives, canonical forms at product and base type guarantee that the evaluated subterm has the shape the evaluation rule needs, and totality of $\hat{p}$ guarantees a result. For case, if the scrutinee is in D the first case rule applies; otherwise it evaluates to a value of sum type, which is $\mathsf{inl}\, u$ or $\mathsf{inr}\, u$ by canonical forms, and the corresponding branch, after substitution, is closed, well typed, and in D. A fixed point always evaluates by E-Fix, so it is not in D. □

The lemma is the reason the book states its liveness hypothesis in the form it does. For a well-typed transformation, "evaluation of $f\, b$ terminates" in Hypothesis 15.8 is exactly the negation of $f\, b \Uparrow$, and there is no third case to exclude. A transform prefix whose function diverges on the bound value is not an error: the configuration simply has no step of that prefix, forever, and the commitment waits. Whether that is acceptable is a property of the function supplied, and the typed calculus can say precisely which functions it is: those for which the dichotomy of the lemma falls on the side of evaluation.

Lemma 7.2 (Determinism)**.** *If $e \Downarrow u$ and $e \Downarrow u'$ then $u = u'$.*

Proof. By induction on the derivation of $e \Downarrow u$. For each form of e there is one rule, except that a primitive has one rule for a constant argument and one for a pair of constants, selected by the shape of the argument's value; case analysis has one rule per injection, selected by the value of the scrutinee; and a pair or injection of values is covered both by E-Val and by its own rule, which give the same result. In each rule the premises are evaluations of subterms or of substitution instances determined by earlier premises, and so are determined by the induction hypothesis, and the conclusion is a function of their results. □

Determinism is what the book uses. Persistence in the proof of eventual disposition, the stability of replay, and the soundness of the π-encoding all rely on the fact that a transformation, once enabled, has one result. Type preservation for closed terms was established in Chapter 14 from the substitution lemmas.

7.6 Termination

Without fix, every well-typed closed term evaluates: this is strong normalization of the simply typed lambda calculus with products and sums, proved by Tait's method of reducibility and cited here without re-derivation. With fix the layer is Turing complete, and some evaluations diverge. The book takes the second option and pays for it explicitly: the eventual-disposition theorem assumes that the particular transformations and verifiers an execution invokes terminate (Hypothesis 15.8 and Theorem 23.1), rather than assuming termination of the layer as a whole. A system that wants the assumption discharged can restrict transformations to the fix-free fragment and inherit it from strong normalization.

7.7 What a history keeps that a normal form discards

A normal form is the end of a computation and says nothing about how it was reached. Many terms reduce to the same value, and once reduced they are indistinguishable. A ledger keeps more, and exactly how much more depends on which of the history equivalences of Chapter 24 the observer uses.

Proposition 7.3 (Transformations are recorded intensionally)**.** *Let f and f' be distinct terms with $f\,b \Downarrow v$ and $f'\,b \Downarrow v$. The histories that differ only in* $\mathsf{Transform}(c, f, b, v)$ *versus* $\mathsf{Transform}(c, f', b, v)$ *are related by $\approx_{\mathrm{st}}$ and not by $\approx_{\mathrm{tr}}$.*

Proof. The state after either record is $\mathsf{transformed}(b, v)$, and the rest of the histories agree. The records differ, so the histories are not trace equivalent. □

The state keeps the result; the ledger keeps which function produced it. An auditor can tell two extensionally equal transformations apart; a state observer cannot; and by Theorem 24.4 no future continuation of the ledger can depend on the difference. Under the opaque reading of Chapter 3 this is all the history keeps: the function applied and its result, not the evaluation that connected them. Under the transparent reading, if it can be established, the kernel would also hold the reduction sequence, and the history would keep the whole derivation of v.

Chapter 8

Combinators, Erasure, and Linear Resource

This chapter asks two structural questions about the value layer and the kernel. How much of the calculus depends on names? And what does the calculus do with resources that are discarded or duplicated? The first question is answered by combinatory logic, the second by linear logic, and the answers fix the resource discipline Part IV builds on.

8.1 Names

Combinatory logic shows that bound variables are dispensable in the value layer. Bracket abstraction translates every lambda term into a term built from the combinators S and K by application alone, and the translation is correct: a term and its translation behave alike under application to arguments. This is standard and is cited without re-derivation. The lambda layer can therefore be presented without variables at all.

A small case shows the mechanism. Bracket abstraction $[x]$ is defined by $[x]x = \mathsf{I}$, where $\mathsf{I} = \mathsf{SKK}$; $[x]M = \mathsf{K}\,M$ when x does not occur in M; and $[x](M\,N) = \mathsf{S}\,([x]M)\,([x]N)$ otherwise. For the first projection of a curried pair, $\lambda x.\lambda y.x$,

$$[y]x = \mathsf{K}\,x, \qquad [x](\mathsf{K}\,x) = \mathsf{S}\,([x]\mathsf{K})\,([x]x) = \mathsf{S}\,(\mathsf{KK})\,\mathsf{I},$$

and on arguments a and b,

$$\mathsf{S}\,(\mathsf{KK})\,\mathsf{I}\,a\,b \;\to\; \mathsf{KK}\,a\,(\mathsf{I}\,a)\,b \;\to\; \mathsf{K}\,(\mathsf{I}\,a)\,b \;\to\; \mathsf{I}\,a \;\to^{*}\; a,$$

where $\mathsf{I}\,a = \mathsf{SKK}\,a \to \mathsf{K}\,a\,(\mathsf{K}\,a) \to a$. The variable has gone, and in its place are two uses of K: one discards the a passed to the outer KK, which is recomputed through I, and one discards b. The next section is about the second.

Commitment identities are not variables, and nothing analogous removes them. A bound variable is a placeholder, and any consistent renaming of it changes nothing. A commitment identity is *generated*: pop must produce a name distinct from every name already in use, and that requirement refers to the current state, not to the structure of a term. Combinatory reduction is a function of terms alone and has no state to consult, so it has nothing that can play the role of a fresh name. Fresh identities are therefore a feature of the process layer, supplied by restriction in the π-calculus (Chapter 12) and by the freshness premise here, and not of the functional core.

8.2 Erasure

The combinator K discards its second argument: $\mathsf{K}\,x\,y \to x$. Under the opaque reading of Chapter 3, such internal discarding is not recorded. A transformation record keeps the function, its input, and its result, so the ledger knows that f was applied to b and gave v, but not which parts of b the computation ignored. Recording erasure inside evaluation would require the transparent reading, in which the kernel holds the reduction itself.

At the level of commitments, discarding is exactly what the calculus forbids, and the linear type discipline is how it forbids it.

8.3 Linear resource

Recall the three kinds of assumption of Chapter 14: rights, which may be neither duplicated nor discarded; tokens, which may be discarded but not duplicated; and facts, which may be used any number of times. In the terminology of substructural logic these are linear, affine, and exponential (!) assumptions. The kernel has its own discipline, and it is a different one.

Proposition 8.1 (The kernel is affine in options)**.** *Kernel admissibility allows each option to be popped at most once and allows options never to be popped. Read as a resource discipline on options, it therefore admits weakening and forbids contraction: it is affine, not linear.*

Proof. The admissibility condition of $\mathsf{Pop}(x)$ in Chapter 3 requires x not already popped, which forbids using an option twice. Nothing requires an option to be popped, which permits leaving it unused. □

The typed disposition calculus strengthens this. A live commitment's right is linear, because T-Nil refuses to let a process end while holding one: a commitment must be advanced, refused, or waited for, never silently dropped. That is the static form of the principle that a refusal must be recorded rather than left implicit. The kernel, which only documents what happens, can afford weakening; a discipline that promises every commitment a disposition cannot.

The absence of weakening is visible in the functions the value layer can express over linear resources.

Proposition 8.2 (No linear discarding)**.** *For distinct atomic types A and B, no closed term of the purely linear fragment, without* !*, has type $A \multimap B \multimap A$.*

Proof. Such a term would be a proof of $A \multimap (B \multimap A)$ in multiplicative linear logic. Every provable sequent of that logic is *balanced*: each atom occurs as often positively as negatively. In $A \multimap (B \multimap A)$ the atom B occurs once, negatively, and never positively. This is a standard property of multiplicative linear logic, cited without re-derivation. □

So the linear analogue of K does not exist: a function cannot consume a right and return without using it. With the exponential the type $A \multimap !B \multimap A$ is inhabited, which is why facts, which may be ignored, are exponential, and rights, which may not, are not.

8.4 The three disciplines together

The calculus thus uses three resource disciplines at once, each where it belongs. Options in the kernel are affine: committed to at most once, possibly never. Rights to advance a commitment are linear: exactly one holder, never dropped. Facts, the authority policy and the terminality of finished commitments, are exponential: consulted freely, never consumed. The ledger itself is outside all three. It is not a resource at all but a record, and it neither shrinks when resources are consumed nor grows when they are duplicated.

Part III

Processes and Labelled Transitions

Chapter 9

Labelled Disposition Semantics

9.1 Actions and records

Observable actions are

$$\alpha ::= \mathsf{pop}(c, q) \mid \mathsf{compensate}(a, c, q, d) \mid \mathsf{bind}(a, c, b) \mid \mathsf{transform}(c, f, b, v) \\ \mid \mathsf{verify}(a, c, v, w) \mid \mathsf{refuse}(a, c, r) \mid \mathsf{collapse}(a, c, k, w, o).$$

Every observable action has a canonical ledger record $\mathrm{record}(\alpha)$, the record of Section 1.4 with the capitalized constructor and the same fields. This pairing is part of the semantics rather than an optional logging policy. Subjects and projections are as defined there.

9.2 Primitive rules

POP introduces a fresh commitment:

$$\frac{c \notin \mathrm{dom}(A)}{\langle \mathsf{pop}\ q\ \mathsf{as}\ c.P, S, A, O, H\rangle \xrightarrow{\mathsf{pop}(c,q)} \langle P, S, A[c \mapsto \mathsf{open}(q)], O, H \cdot \mathsf{Pop}(c, q)\rangle}.$$

A compensating pop introduces a fresh successor to a terminal commitment:

$$\frac{c \notin \mathrm{dom}(A) \qquad \mathsf{terminal}(A(d)) \qquad \mathsf{may}(a, \mathsf{compensate}, d)}{\langle \mathsf{pop}_a\ q\ \mathsf{as}\ c\ \mathsf{after}\ d.P, S, A, O, H\rangle \xrightarrow{\mathsf{compensate}(a,c,q,d)} \langle P, S, A[c \mapsto \mathsf{open}(q)], O, H \cdot \mathsf{Compensates}(a, c, q, d)\rangle}.$$

Binding advances an open commitment while preserving its identity:

$$\frac{A(c) = \mathsf{open}(q) \qquad \mathsf{may}(a, \mathsf{bind}, c)}{\langle \mathsf{bind}_a\ c\ \mathsf{to}\ b.P, S, A, O, H\rangle \xrightarrow{\mathsf{bind}(a,c,b)} \langle P, S, A[c \mapsto \mathsf{bound}(q, b)], O, H \cdot \mathsf{Bind}(a, c, b)\rangle}.$$

Transformation evaluates a lambda-layer function and records both its input and output:

$$\frac{A(c) = \mathsf{bound}(q, b) \qquad f\, b \Downarrow v}{\langle \mathsf{transform}\ c\ \mathsf{with}\ f.P, S, A, O, H\rangle \xrightarrow{\mathsf{transform}(c,f,b,v)} \langle P, S, A[c \mapsto \mathsf{transformed}(b, v)], O, H \cdot \mathsf{Transform}(c, f, b, v)\rangle}.$$

Verification runs the verifier and branches on its result. Write

$$V \equiv \mathsf{verify}_a\ c\ \mathsf{using}\ g\ \mathsf{then}\ P\ \mathsf{else}\ Q.$$

The successful branch produces an indexed witness:

$$\frac{A(c) = \mathsf{transformed}(b, v) \qquad \mathsf{may}(a, \mathsf{verify}, c) \qquad g(c, v) \Downarrow \mathsf{ok}(w)}{\langle V, S, A, O, H\rangle \xrightarrow{\mathsf{verify}(a,c,v,w)} \langle P, S, A[c \mapsto \mathsf{verified}(v, w)], O, H \cdot \mathsf{Verify}(a, c, v, w)\rangle}.$$

The failing branch records the failure certificate and leaves the commitment live:

$$\frac{A(c) = \mathsf{transformed}(b, v) \qquad \mathsf{may}(a, \mathsf{verify}, c) \qquad g(c, v) \Downarrow \mathsf{fail}(\varphi)}{\langle V, S, A, O, H\rangle \xrightarrow{\mathsf{verificationFailed}(a,c,v,\varphi)} \langle Q, S, A[c \mapsto \mathsf{verificationFailed}(v, \varphi)], O, H \cdot \mathsf{VerificationFailed}(a, c, v, \varphi)\rangle}.$$

Explicit refusal is available from any live state d:

$$\frac{A(c) = d \qquad \mathsf{live}(d) \qquad \mathsf{may}(a, \mathsf{refuse}, c)}{\langle \mathsf{refuse}_a\ c\ \mathsf{because}\ r.P, S, A, O, H\rangle \xrightarrow{\mathsf{refuse}(a,c,r)} \langle P, S, A[c \mapsto \mathsf{refused}(r)], O, H \cdot \mathsf{Refuse}(a, c, r)\rangle}.$$

Collapse commits to an observation of a verified commitment, using the witness held in its state:

$$\frac{A(c) = \mathsf{verified}(v, w) \qquad \mathsf{may}(a, \mathsf{collapse}, c) \qquad o = \mathsf{observe}_k(v)}{\langle \mathsf{collapse}_{a,k}\ c.P, S, A, O, H\rangle \xrightarrow{\mathsf{collapse}(a,c,k,w,o)} \left\langle \begin{matrix} P, S, A[c \mapsto \mathsf{collapsed}(o)], O \cup \{(c, k, o)\}, \\ H \cdot \mathsf{Collapse}(a, c, k, w, o) \end{matrix} \right\rangle}.$$

The subscript k names the observation rule, and the record carries it. Collapse is therefore not one universal quotient imposed on every history. Different applications may expose different distinctions, but every use declares which rule did the identifying. The observable label of a collapse is the pair $\langle k, o\rangle$, not the payload o alone, so equal payloads produced under different rules remain distinct: $\langle k_1, o\rangle \neq \langle k_2, o\rangle$ when $k_1 \neq k_2$. An observer who wants to ignore the rule may project it away, but that projection is itself a declared collapse rule, not a default. Observations are stored as (c, k, o), so two commitments collapsing to equal values also remain distinguishable in O.

External obligations are discharged by the environment, with authenticated evidence:

$$\frac{\mathsf{live}(A(c)) \qquad \mathsf{may}(a, \mathsf{discharge}, \omega) \qquad \mathsf{checkCert}(\varphi, c, \omega, \mathsf{discharged})}{\langle \mathsf{await}_a\ c\ \omega.P, S, A, O, H\rangle \xrightarrow{\mathsf{discharge}(a,c,\omega,\varphi)} \langle P, S, A, O, H \cdot \mathsf{Discharged}(a, c, \omega, \varphi)\rangle}.$$

For the deadline form, discharge takes the first branch by the same rule, and a certified timeout takes the second:

$$\frac{\mathsf{live}(A(c)) \qquad \mathsf{may}(b, \mathsf{clock}, \omega) \qquad \mathsf{checkCert}(\varphi, c, \langle \omega, t\rangle, \mathsf{expired})}{\langle \mathsf{await}_a\ c\ \omega.P\ \mathsf{within}_b\ t\ \mathsf{else}\ Q, S, A, O, H\rangle \xrightarrow{\mathsf{timeout}(b,c,\omega,t,\varphi)} \langle Q, S, A, O, H \cdot \mathsf{TimedOut}(b, c, \omega, t, \varphi)\rangle}.$$

The discharge record must be classified, since the book has just argued that no fifth causal primitive is needed. It is *kernel-derived*: a guarded $\mathsf{Bind} \circ \mathsf{Pop}$ of an evidence-carrying proposal to the history of c, the same shape as binding, transformation, and verification (Chapter 3), differing from them only in leaving the lifecycle state where it was. It is not a primitive and not a mere annotation, because it changes what the process may do next. Three nearby things are *not* ledger events and must stay distinct from it: a pending obligation, which is process state (the await prefix is simply still there); a denied attempt to discharge, which belongs at most on the annotation channel; and nonresponse, which is the absence of any discharge. None of these ends the commitment. If a pending obligation is to end it, an authorized principal must refuse it explicitly. A timeout record is classified the same way as a discharge: kernel-derived, lifecycle-neutral, and authenticated.

The stability hypothesis of Section 1.4 applies to the validity predicate, failure certificates, and observation maps used in these rules.

9.3 Contextual rules

The primitive rules act on a prefix at the top of the process. Parallel components share the store, commitment map, observation set, and history, so a step of one component is a step of the whole:

$$\frac{\langle P, S, A, O, H\rangle \xrightarrow{\alpha} \langle P', S', A', O', H'\rangle}{\langle P \mid Q, S, A, O, H\rangle \xrightarrow{\alpha} \langle P' \mid Q, S', A', O', H'\rangle} \qquad \frac{\langle P, S, A, O, H\rangle \xrightarrow{\alpha} \langle P', S', A', O', H'\rangle}{\langle P + Q, S, A, O, H\rangle \xrightarrow{\alpha} \langle P', S', A', O', H'\rangle}$$

$$\frac{\langle P, S, A, O, H\rangle \xrightarrow{\alpha} \langle P', S', A', O', H'\rangle}{\langle (\nu c)P, S, A, O, H\rangle \xrightarrow{\alpha} \langle (\nu c)P', S', A', O', H'\rangle} \qquad \frac{P \equiv P_1 \quad \langle P_1, \ldots\rangle \xrightarrow{\alpha} \langle P_2, \ldots\rangle \quad P_2 \equiv P'}{\langle P, \ldots\rangle \xrightarrow{\alpha} \langle P', \ldots\rangle}$$

Choice is resolved by whichever summand acts first. The unchosen summand is neither evaluated nor refused: the rule for $P + Q$ requires a transition of P only, inspects no premise of Q, and records nothing about Q. Internal choice resolution is therefore a different thing from disposition refusal, and nothing about a discarded branch can later be read off the ledger.

Remark 9.1 (Restriction scopes naming, not visibility). In the π-calculus, restriction makes a name private and hides actions on it from the environment. Here the restriction rule places no side condition on the label, and the record is appended to a history shared by every component. Restriction here is lexical scope, not confidentiality: it limits which parts of a process can *name* a commitment, and so which parts can act on it, but not who can *see* what happened to it. The ledger is not scoped. Whether this makes restriction behave like a capability discipline is a question for Chapter 12, not a claim of this one.

9.4 Worked derivations

The successful route. Let

$$\begin{aligned} P_0 = {} & \mathsf{pop}\ q\ \mathsf{as}\ c.\ \mathsf{bind}_a\ c\ \mathsf{to}\ b.\ \mathsf{transform}\ c\ \mathsf{with}\ f. \\ & \mathsf{verify}_{a'}\ c\ \mathsf{using}\ g\ \mathsf{then}\ \mathsf{collapse}_{a'',k}\ c.\mathbf{0}\ \mathsf{else}\ \mathbf{0}, \end{aligned}$$

with $f\,b \Downarrow v$, $g(c, v) \Downarrow \mathsf{ok}(w)$, $o = \mathsf{observe}_k(v)$, and each principal holding the authority its step requires. From the empty configuration the five primitive rules fire in order:

$$C_0 \xrightarrow{\mathsf{pop}(c,q)} C_1 \xrightarrow{\mathsf{bind}(a,c,b)} C_2 \xrightarrow{\mathsf{transform}(c,f,b,v)} C_3 \xrightarrow{\mathsf{verify}(a',c,v,w)} C_4 \xrightarrow{\mathsf{collapse}(a'',c,k,w,o)} C_5,$$

with

i	$A_i(c)$	H_i
1	$\mathsf{open}(q)$	$\langle \mathsf{Pop}(c, q)\rangle$
2	$\mathsf{bound}(q, b)$	$H_1 \cdot \mathsf{Bind}(a, c, b)$
3	$\mathsf{transformed}(b, v)$	$H_2 \cdot \mathsf{Transform}(c, f, b, v)$
4	$\mathsf{verified}(v, w)$	$H_3 \cdot \mathsf{Verify}(a', c, v, w)$
5	$\mathsf{collapsed}(o)$	$H_4 \cdot \mathsf{Collapse}(a'', c, k, w, o)$

and $O_5 = \{(c, k, o)\}$. Each history is a proper prefix of the next. The premise of each rule is visible in the previous row: the bind rule needs the open state of row 1, the transform rule reads the b of row 2, and so on.

Refusal at every live stage. The four refusal routes are not redundant, because they establish that the refusal rule covers every live state, the failed state included.

1. *After pop.* From C_1, $\mathsf{refuse}_a\ c\ \mathsf{because}\ r$ fires with $A_1(c) = \mathsf{open}(q)$, giving $H_1 \cdot \mathsf{Refuse}(a, c, r)$.
2. *After binding.* From C_2, a reviewer with refusal authority judges the bound value unsuitable; the explicit rule fires from $\mathsf{bound}(q, b)$.
3. *After an inadmissible transformation.* From C_3, the explicit rule fires from $\mathsf{transformed}(b, v)$ with a reason stating why v is unacceptable.
4. *After negative verification.* From C_3, if $g(c, v) \Downarrow \mathsf{fail}(\varphi)$, the failing verification rule fires and the process continues with its else branch, giving $H_3 \cdot \mathsf{VerificationFailed}(a', c, v, \varphi)$ with c live in $\mathsf{verificationFailed}(v, \varphi)$. The else branch then contains the refusal, $\mathsf{refuse}_h\ c\ \mathsf{because}\ r$, by a principal h holding refusal authority, and the explicit refusal rule fires from the failed state.

Remark 9.2 (Denial of an actor is not refusal of a commitment). One tempting fifth route is "refusal because the binding was unauthorized". There is no such rule, and there should not be:

$$\neg\mathsf{may}(a, \mathsf{bind}, c) \not\Rightarrow \mathsf{Refused}(c).$$

If an unauthorized attempt refused c, authority failure would become an ambient cancellation capability: any principal could end any commitment by attempting something it is not entitled to do. Three outcomes must be kept apart.

- $\mathsf{DeniedAttempt}(a, c, \alpha)$: a principal tried an act it may not perform. Nothing about c changes. If recorded at all, it belongs on the kernel's non-causal annotation channel, an audit plane beside the ledger, where it supports security analysis without becoming a causal event.
- $\mathsf{Blocked}(c, \omega)$: the commitment cannot advance until an external obligation ω is discharged. It is a positive runtime form, introduced in Chapter 15, and it changes nothing about the lifecycle either.
- $\mathsf{Refused}(b, c, r)$: an authorized principal b ended the commitment for reason r. Only this outcome changes the lifecycle.

In the present calculus an unauthorized attempt simply has no transition. In the typed calculus it cannot even be written, since every attributed prefix requires its authority as a typing premise.

Independent steps commute. Let $A(c) = \mathsf{open}(q)$ and $A(c') = \mathsf{open}(q')$ with $c \neq c'$, and let the process be $\mathsf{bind}_a\ c\ \mathsf{to}\ b.\mathbf{0} \mid \mathsf{bind}_{a'}\ c'\ \mathsf{to}\ b'.\mathbf{0}$. Either component may act first, by the parallel rule (with commutativity of $\mid$ for the right component). The two interleavings reach the same process, commitment map, and observation set, and histories

$$H \cdot \mathsf{Bind}(a, c, b) \cdot \mathsf{Bind}(a', c', b') \quad \text{and} \quad H \cdot \mathsf{Bind}(a', c', b') \cdot \mathsf{Bind}(a, c, b),$$

which differ by one exchange of independent records. In general:

Proposition 9.3 (Diamond for independent steps)**.** *Let $C = \langle P_1 \mid P_2, S, A, O, H\rangle$, and suppose P_1 can perform α and P_2 can perform β from C, with* record(α) *and* record(β) *independent in the sense of Definition 20.9. Then both interleavings are possible and reach configurations equal in every component except the history, where they differ by exchanging the two records.*

Proof. The premises of each rule read the commitment map only at the subject of its record, and, for compensation, at its predecessor. By independence, the other step writes neither key (Theorem 9.5), so each premise still holds after the other step, and each update produces the same value in either order. The process components evolve separately. Observation sets are joined by union. A pop and a compensating pop each require their subject to be fresh; independence gives distinct subjects, so neither introduction uses the other's identity. □

The hypothesis that the two steps come from different parallel components matters. In $\mathsf{bind}_a\ c\ \mathsf{to}\ b.\,\mathsf{bind}_{a'}\ c'\ \mathsf{to}\ b'.\mathbf{0}$ the records are still independent, but only one order is possible, because the prefix imposes it. Independence of records says replay does not care about their order; it does not say the program allows both.

Competing terminal steps. Let $A(c) = \mathsf{verified}(v, w)$ and let the process be $\mathsf{refuse}_a\ c\ \mathsf{because}\ r.\mathbf{0} \mid \mathsf{collapse}_{a',k}\ c.\mathbf{0}$. Either component may fire first. If the refusal fires, c becomes $\mathsf{refused}(r)$ and the collapse rule's premise $A(c) = \mathsf{verified}(v, w)$ fails. If the collapse fires, c becomes $\mathsf{collapsed}(o)$, which is not live, and the refusal rule's premise fails. Exactly one terminal record is appended.

This settles a point about concurrency. Terminal exclusivity does not rest on the order in which rules are written; it rests on the commitment map being shared, so that the first terminal step falsifies the premise of every competitor. That in turn rests on an assumption the interleaving semantics makes silently.

Hypothesis 9.4 (Atomic commitment map)**.** *Each transition reads the commitment map, checks its premises, and updates the map as one indivisible step.*

Under this hypothesis exclusivity is *operational*. In a distributed implementation the hypothesis must be earned: without a sequencer, a compare-and-swap, a consensus object, or some other linearization point, two replicas could each read a live state, each append a terminal record, and produce incompatible dispositions that no later replay can reconcile. Chapter 22 states what each mechanism provides. There are therefore two grades of the result: operational exclusivity, which holds for every execution under Hypothesis 9.4, and static exclusion of competitors, which holds for well-typed programs. What the untyped calculus does not do is say anything about the loser, which is left with a prefix it can never perform and no record of why. The linear discipline of Part IV turns that situation into a static error: a well-typed program cannot hold two rights to advance the same commitment, so it cannot contain both components in the first place.

9.5 First preservation results

Theorem 9.5 (Commitment identity)**.** *If $C \xrightarrow{\alpha} C'$, then* $\mathrm{dom}(A) \subseteq \mathrm{dom}(A')$*, and*

$$\mathrm{dom}(A') \setminus \mathrm{dom}(A) = \begin{cases} \{c\} & \text{if } \alpha = \mathsf{pop}(c, q) \text{ or } \alpha = \mathsf{compensate}(a, c, q, d), \\ \varnothing & \text{otherwise,} \end{cases}$$

with $c \notin \mathrm{dom}(A)$ *in the first case. Moreover* $A'(e) = A(e)$ *for every* $e \neq \mathrm{subj}(\mathrm{record}(\alpha))$.

Proof. By induction on the derivation. Each primitive rule updates A at no more than one key, the subject of its record; the discharge and timeout rules write back the value already there. The two introducing rules have $c \notin \mathrm{dom}(A)$ as a premise and add that key; every other primitive rule has a premise $A(c) = \ldots$ and so updates an existing key. The contextual rules pass A and A' through unchanged from their premise. □

Since the domain never shrinks, a commitment identity once introduced remains in the map for the rest of the execution. The spine's provision for an archival operation that preserves identity in the ledger is therefore not needed in the present calculus; it becomes relevant only if a later extension removes terminal commitments from A for efficiency.

Theorem 9.6 (Lifecycle monotonicity)**.** *If* $C \to C'$ *and* c *occurs in both commitment maps, then*

$$A(c) \preceq_D A'(c).$$

In particular, no ordinary transition reopens a refused or collapsed commitment.

Proof. By cases on the primitive rule at the root of the derivation, the contextual rules transmitting the property. The introducing rules concern a fresh identifier and, by Theorem 9.5, leave every existing key unchanged; in particular a compensating pop does not alter its predecessor. Each advancing rule follows one generating edge of $\preceq_D$. Both refusing rules follow the refusal edge from a live state. Neither terminal state satisfies the premise of an advancing or refusing rule. □

Theorem 9.7 (History monotonicity)**.** *If* $C \xrightarrow{\alpha} C'$*, then*

$$H' = H \cdot \mathrm{record}(\alpha),$$

and hence $H \sqsubseteq H'$.

Proof. Inspection of the primitive rules shows that each appends its paired record. The contextual rules pass the history through unchanged from their premise, and structural congruence acts on the process component alone. No rule has a premise or conclusion in which a history suffix is removed or replaced. □

Corollary 9.8 (Terminal exclusivity, map form)**.** *Along any execution, no commitment enters both* refused *and* collapsed.

Proof. Whichever terminal state is entered first, lifecycle monotonicity keeps the commitment in that state at every later configuration, since $\preceq_D$ relates each terminal state only to itself. By Theorem 9.5 the identity cannot be removed and reintroduced in a different state. □

The history form of this corollary, which is what an auditor reading the ledger needs, is Corollary 18.5.

9.6 Authority

Define the authority an action requires, $\mathsf{auth}(\alpha)$, by

$$\mathsf{auth}(\mathsf{bind}(a,c,b)) = \mathsf{bind}, \qquad \mathsf{auth}(\mathsf{verify}(a,c,v,w)) = \mathsf{verify},$$
$$\mathsf{auth}(\mathsf{collapse}(a,c,k,w,o)) = \mathsf{collapse}, \quad \mathsf{auth}(\mathsf{compensate}(a,c,q,d)) = \mathsf{compensate},$$

and

$$\mathsf{auth}(\mathsf{refuse}(a,c,r)) = \mathsf{refuse}, \qquad \mathsf{auth}(\mathsf{verificationFailed}(a,c,v,\varphi)) = \mathsf{verify},$$

$\mathsf{auth}(\mathsf{discharge}(a,c,\omega,\varphi)) = \mathsf{discharge}$, and $\mathsf{auth}(\mathsf{timeout}(b,c,\omega,t,\varphi)) = \mathsf{clock}$. The object of the required authority is the action's subject, except for compensation, where it is the predecessor d, and discharge and timeout, where it is the obligation ω.

Theorem 9.9 (Authorized transition)**.** *If $C \xrightarrow{\alpha} C'$ and α is attributed to principal a, then $\mathsf{may}(a, \mathsf{auth}(\alpha), x)$ held in C, where x is the object of the required authority.*

Proof. By inversion on the derivation. The contextual rules preserve the label, so the root primitive rule determines it, and every attributed primitive rule carries the may premise for exactly the kind auth assigns to its label. In particular a refusal label comes only from the explicit refusal rule, whose premise is refusal authority, and a verification-failure label only from the failing verification rule, whose premise is verification authority. □

Two different guarantees concern verification failures, and each has its own scope.

Proposition 9.10 (Source authenticity)**.** *If $\mathsf{VerificationFailed}(a,c,v,\varphi)$ occurs in a history produced by execution, then at the configuration where it was appended, c was $\mathsf{transformed}(b,v)$ for some b, principal a held verification authority over c, and the verifier named in the process returned $\mathsf{fail}(\varphi)$ on (c,v).*

Proof. By history monotonicity the record was appended by a single transition, which can only have been an instance of the failing verification rule, whose premises are the statement. □

This is a statement about programs. It says nothing about a ledger that arrives from elsewhere, since someone with write access to storage could append a record directly, without running any process. For that setting the guarantee comes from replay.

Proposition 9.11 (Ledger acceptance)**.** *If $\delta^*(H|_c)$ is defined and $H|_c$ contains the record $\mathsf{VerificationFailed}(a,c,v,\varphi)$, then the preceding record of $H|_c$ left c in a state $\mathsf{transformed}(b,v)$ for which*

$$\mathsf{checkCert}(\varphi, c, v, \mathsf{fail})$$

holds. Under Hypothesis 1.5, φ was issued by the verifier it names, at the version it names, on that commitment and value.

Proof. The only clause of δ accepting the record is the verification-failure clause, whose incoming state and side condition are the statement. The second sentence is the authentication half of Hypothesis 1.5. □

What replay cannot check is that the principal a in the record held verification authority; that requires the policy, which an auditor can supply alongside the ledger. And the proposition is only as strong as the certificate scheme: if certificates can be forged, so can ledger-level verification failures. Neither proposition concerns refusal at all. A verification failure is evidence; whether it leads to refusal is decided by a separate, separately authorized act.

In the present calculus the policy judgement may is fixed, so the spine's non-amplification corollary holds trivially: no transition changes what any principal may do. It becomes a substantive result once delegation is added in Chapter 16, where authority can grow, and the claim is that it grows only through a delegation record.

Chapter 10

CCS: Synchronization and Binding

The calculus of communicating systems is organized around one idea: two agents synchronize when one performs an action a and the other its complement $\bar{a}$, and the pair becomes a single internal step τ. The disposition calculus has an operator called binding, and a natural first guess is that binding is its synchronization. This chapter shows that the guess is wrong, and in a strong sense: in the typed calculus, parallel components do not interact at all. That fact yields a separation result in one direction and a forward correspondence in the other.

10.1 Erasure into CCS

Let the erasure $\mathcal{E}$ send a disposition process to a CCS process over an alphabet of actions that record the kind of each act and its subject but nothing else:

$$\begin{aligned} \mathcal{E}(\mathbf{0}) &= \mathbf{0}, & \mathcal{E}(\pi.P) &= \lambda(\pi).\mathcal{E}(P), \\ \mathcal{E}(P \mid Q) &= \mathcal{E}(P) \mid \mathcal{E}(Q), & \mathcal{E}(P + Q) &= \mathcal{E}(P) + \mathcal{E}(Q), \\ \mathcal{E}((\nu c)P) &= \mathcal{E}(P), & \mathcal{E}(!P) &= !\,\mathcal{E}(P), \end{aligned}$$

with the two-branch forms sent to sums,

$$\begin{aligned} \mathcal{E}(\mathsf{verify}_a\ c\ \mathsf{using}\ g\ \mathsf{then}\ P\ \mathsf{else}\ Q) &= \mathsf{ver}_c^{+}.\mathcal{E}(P) + \mathsf{ver}_c^{-}.\mathcal{E}(Q), \\ \mathcal{E}(\mathsf{await}_a\ c\ \omega.P\ \mathsf{within}_b\ t\ \mathsf{else}\ Q) &= \mathsf{dis}_c.\mathcal{E}(P) + \mathsf{to}_c.\mathcal{E}(Q), \end{aligned}$$

and λ giving pop, comp_d, bind_c, trans_c, refuse_c, $\mathsf{collapse}_c$, or dis_c for the other prefixes. No action has a complement, so no two erased components ever synchronize, and CCS restriction, which matters only for complementary actions, is not needed. The same map sends each disposition label α to $\lambda(\alpha)$ by forgetting its annotations.

Proposition 10.1 (Forward correspondence)**.** *If* $\langle P, S, A, O, H\rangle \xrightarrow{\alpha} \langle P', S', A', O', H'\rangle$ *then* $\mathcal{E}(P) \xrightarrow{\lambda(\alpha)} \equiv\ \mathcal{E}(P')$.

Proof. By induction on the transition derivation. Each primitive rule fires a prefix whose erasure is $\lambda(\alpha)$ followed by the erasure of its continuation; for pop, the fresh name chosen by the rule is an α-variant of the bound one. The parallel, sum, and restriction rules lift directly, and the congruence rule uses the fact that $\mathcal{E}$ maps each structural law of Chapter 5 to a structural law of CCS. □

The converse fails, and the way it fails is instructive.

Proposition 10.2 (What the erasure forgets)**.** *(a) $\mathcal{E}$ is not sound: there is a configuration with no transition whose erasure has one.*

(b) $\mathcal{E}$ is not fully abstract for any equivalence on configurations that distinguishes refusal reasons or acting principals: there are configurations whose processes have identical erasures and whose executions produce histories that differ.

Proof. (a) With $A(c) = \mathsf{open}(q)$, the process $\mathsf{collapse}_{a,k}\ c.\mathbf{0}$ has no transition, while its erasure $\mathsf{collapse}_c.\mathbf{0}$ does. The erasure forgets the commitment map, which is what gates enabledness. (b) From the same configuration, $\mathsf{refuse}_a\ c\ \mathsf{because}\ r_1.\mathbf{0}$ and $\mathsf{refuse}_b\ c\ \mathsf{because}\ r_2.\mathbf{0}$ both erase to $\mathsf{refuse}_c.\mathbf{0}$, and their executions end in $\mathsf{refused}(r_1)$ and $\mathsf{refused}(r_2)$ with records naming different principals. □

So CCS captures the shape of disposition behaviour, the order and branching of kinds of act, and forgets both what gates it and what it records.

10.2 Typed components do not interact

The converse direction turns on a fact about the typed calculus that the book has used without naming.

Lemma 10.3 (Pruning)**.** *Let $C[X, Y]$ be a closed well-typed configuration built from a context C with two holes, filled by X and Y, and suppose $C[X, Z]$ is well typed too. If ξ is an execution of $C[X, Y]$, then deleting from ξ every step fired by a prefix originating in Y leaves an execution of $C[X, Z]$.*

Proof. Every prefix in the term $C[X, Y]$ originates in exactly one of C, X, and Y, and descendants of a prefix's continuation originate where the prefix did. We check that each remaining step is still enabled, in order. Its lifecycle and data premises concern commitments whose rights its own occurrence holds (Lemma 15.1 and condition (C3)); rights live in the text of one occurrence and are never transferred, so no step from Y ever wrote those commitments. Its freshness premise is satisfied by the same name, since deleting steps only shrinks the set of names in use. Its authority premise is a fixed fact, or an exclusive capability held in its own text. A terminal fact in its typing context comes either from the initial configuration or from a terminal step in its own text, so a compensation premise does not depend on Y. Evaluation and verification are deterministic. A step from Y that resolved a sum discarded the other summands, so no remaining step lies in them; after the deletion the sum is simply unresolved. Finally, the deleted records are exactly the records of commitments owned by Y's occurrence, together with records appended after the context handed Y a right, so every remaining projection is an unchanged path or a prefix of one, and replay remains defined. In $C[X, Z]$ the hole occupied by Z never steps. □

The form in which pruning is used is a decomposition of contexts. Write $D \Downarrow$ when some execution of D appends a record of a designated success kind.

Corollary 10.4 (Context decomposition)**.** *Let C be a two-hole context, and let X, Y be such that $C[X, Y]$ is closed and well typed. If $C[X, Y] \Downarrow$, then*

$$C[X, Z] \Downarrow \text{ for every } Z \text{ with } C[X, Z] \text{ well typed,}$$
$$\text{or}$$
$$C[W, Y] \Downarrow \text{ for every } W \text{ with } C[W, Y] \text{ well typed.}$$

This holds for every context, including contexts that allocate commitments by pop *and distribute their rights to the holes, and contexts that place the holes under sums or replication.*

Proof. Take a successful execution ξ of $C[X, Y]$ and the prefix that appended the success record; it originates in exactly one of C, X, and Y. If it originates in C or X, delete from ξ every step originating in Y; by Lemma 10.3 the result is an execution of $C[X, Z]$ for any well-typed choice of Z, and it keeps the success record, which no deleted step appended. If it originates in Y, delete every step originating in X and apply the lemma with the roles of the holes exchanged. The lemma's proof already covers contexts that allocate commitments and hand rights to a hole, since a right handed to a hole is held in that hole's text from then on, and the records written with it are among those deleted when that hole is pruned. A sum in the context that the execution resolved into a hole is resolved the same way in the pruned execution, because the resolving step originates in the hole that is kept or in the context, never in the pruned hole. □

The disjunction is the whole point. Success in a composite never needs both holes: one of them can always be replaced by anything, and the composite still succeeds. That is the opposite of synchronization, where success needs both partners.

Pruning says that a well-typed component's behaviour never depends on another's. Two components can race for nothing, pass nothing, and wait for nothing but the environment. That is the formal content of the answer to this chapter's opening question: binding couples a commitment to a value; it does not couple two agents.

10.3 A separation

Consider CCS with a success action ✓, in the reduction semantics Gorla uses, where $S \Downarrow$ means that S reaches a term that can perform ✓ by internal steps only. For the target take closed well-typed configurations, with success meaning that some record of a designated kind, say a collapse under a rule $k_{\checkmark}$, can be appended.

Theorem 10.5 (No compositional encoding of synchronization)**.** *There is no translation from CCS into closed well-typed disposition configurations that is compositional and success sensitive.*

Proof. Suppose $[\![\cdot]\!]$ is such a translation and let C be its context for parallel composition at the name set $\{a\}$. The three source terms

$$a.\checkmark \mid \bar{a}.\mathbf{0}, \qquad a.\checkmark \mid \mathbf{0}, \qquad \mathbf{0} \mid \bar{a}.\mathbf{0}$$

all have free names $\{a\}$, so their translations are $C[[\![a.\checkmark]\!], [\![\bar{a}.\mathbf{0}]\!]]$, $C[[\![a.\checkmark]\!], [\![\mathbf{0}]\!]]$, and $C[[\![\mathbf{0}]\!], [\![\bar{a}.\mathbf{0}]\!]]$. The first source term succeeds, after one synchronization; the other two do not, since neither has any internal step. By success sensitiveness the first translation succeeds. By

Corollary 10.4, either $C[[\![a.\checkmark]\!], Z]$ succeeds for every well-typed Z, in particular for $Z = [\![\mathbf{0}]\!]$, or $C[W, [\![\bar{a}.\mathbf{0}]\!]]$ succeeds for every well-typed W, in particular for $W = [\![\mathbf{0}]\!]$. The translations of the second and third source terms are well typed, being translations, so either way a translation of a term that does not succeed succeeds, contradicting success sensitiveness. □

The theorem needs neither operational correspondence nor name invariance. Compositionality and success sensitiveness suffice, because what the typed calculus lacks is not a particular construct but interaction itself. Two remarks limit its scope.

First, it concerns the typed calculus. In the untyped calculus components can interact through the shared commitment map: if one component holds $\mathsf{bind}_a\ c\ \mathsf{to}\ b$ and another holds $\mathsf{transform}\ c\ \mathsf{with}\ f$, the second is enabled only after the first acts. This is a one-shot, asynchronous, one-way signal, closer to a write-once variable than to a handshake, and whether it suffices to encode asynchronous CCS with single-use channels is listed in Appendix E. The typing discipline removes it deliberately, since it is exactly the cross-component dependence that makes ownership, persistence, and eventual disposition provable.

Second, it concerns closed configurations. Interaction through the environment, one component's act prompting an outside party to discharge another component's obligation, is outside the calculus by design.

10.4 Outcome

The comparison with CCS is a separation in one direction. CCS synchronization cannot be compositionally encoded in the typed disposition calculus (Theorem 10.5), and the natural erasure the other way is complete but neither sound nor fully abstract (Propositions 10.1 and 10.2). With finitely many values, each commitment is a finite-state machine and could be represented as a CCS agent synchronizing with the prefixes that act on it, which would give a sound translation in the other direction; that construction is standard in outline and is not carried out here. The honest summary is that the two calculi answer different questions. CCS is a calculus of communication. The typed disposition calculus deliberately has none, and its contribution lies in what it records about non-communicating acts.

Chapter 11

CSP and the Two Refusals

The failures model of CSP gives the word "refusal" a precise technical meaning, and it is a different meaning from the one this book uses. This chapter makes the difference exact. The result is the weakened form of the second comparative claim, now proved: recorded refusal is not represented by the standard failures model unless an event is added to carry it, and the two notions of refusal classify different objects.

11.1 The failures model

For a process with a labelled transition system over visible events Σ and internal τ, a state is *stable* when it has no τ transition. A pair (s, X), with s a trace of visible events and $X \subseteq \Sigma$, is a *failure* of the process if it can perform s and reach a stable state in which no event of X is possible. The failures model identifies processes with the same traces, failures, and divergences. A CSP refusal is thus a property of a stable state relative to a set of offered events: an observation, made from outside, of what the process cannot do next.

A recorded refusal is a different kind of thing. It is an event in a commitment's history, appended by an authorized principal for a stated reason, and it ends the commitment.

11.2 Reading disposition processes in CSP

Erase a disposition process into CSP as in Chapter 10, with one refinement. The disposition calculus distinguishes two sources of choice. When a principal's process holds several enabled prefixes, which one fires is up to that process, not to anyone outside it; that is internal choice, $\sqcap$. When a process awaits an obligation under a deadline, which branch is taken depends on the environment, discharge or certified expiry; that is external choice, $\Box$. So

$$\mathcal{E}(P + Q) = \mathcal{E}(P) \sqcap \mathcal{E}(Q), \qquad \mathcal{E}(\mathsf{await} \ldots \mathsf{within} \ldots) = \mathsf{dis}_c \to \mathcal{E}(P) \,\Box\, \mathsf{to}_c \to \mathcal{E}(Q),$$

and the two outcomes of verification are an internal choice, since the verifier decides. Every other prefix becomes a visible event, as before. Let R be the set of refusal events refuse_c.

11.3 The two refusals come apart

Proposition 11.1 (Refusal sets and refusal records are independent)**.** *Let* $A(c) = \mathsf{open}(q)$.

(a) The process $\mathsf{refuse}_a\ c\ \mathsf{because}\ r.\mathbf{0}$ *records a refusal of* c*, yet its erasure has no initial failure refusing* refuse_c*: the only initial stable state offers it.*

(b) The untyped process $\mathbf{0}$ *never records anything about* c*, yet its erasure has the failure* $(\langle\rangle, \Sigma)$*: it stably refuses* bind_c *and every other event.*

Proof. Both are immediate from the definition of failures. In (a) the initial state is stable and has the transition refuse_c, so no failure with the empty trace contains that event. In (b) the erasure is $STOP$. □

A CSP refusal can hold where no refusal was recorded, and a refusal can be recorded where CSP reports none. In the failures model the recorded refusal appears, if at all, as the visible event refuse_c, an ordinary event in a trace. That is instrumentation: the event had to be added to the alphabet to be seen.

Theorem 11.2 (Hiding removes the record)**.** *Let* $P_1 = \mathsf{refuse}_a\ c\ \mathsf{because}\ r.\mathbf{0}$ *and* $P_2 = \mathbf{0}$*, both run from a configuration with* $A(c) = \mathsf{open}(q)$*. Then* $\mathcal{E}(P_1) \setminus \mathsf{R}$ *and* $\mathcal{E}(P_2)$ *are equivalent in the failures-divergences model, while the histories their executions produce differ in exactly one refusal record.*

Proof. $\mathcal{E}(P_1) \setminus \mathsf{R}$ performs one internal step and stops; its initial state is unstable, so its only failures are those of the stopped state, $(\langle\rangle, X)$ for every X, and it has no divergence. That is exactly the failures-divergences semantics of $STOP = \mathcal{E}(P_2)$. The execution of P_1 appends $\mathsf{Refuse}(a, c, r)$; P_2 appends nothing. □

So the failures model, without an added event, identifies a configuration in which a commitment was refused, by whom and why, with one in which nothing happened. Its refusal sets have no slot for the record, because they describe what a stable state cannot do, not what was decided. The same example extends to a sequence: a certified verification failure followed by a separately authorized refusal is, after hiding, again indistinguishable from silence, although the ledger holds both the evidence and the decision.

11.4 Declining is not refusing

Proposition 11.1(b) used an untyped process, and for a reason. In the typed calculus a process holding a live right cannot be $\mathbf{0}$ (T-Nil); it can only act on the commitment or wait, and a wait names its obligation and who is expected to discharge it. The typed counterpart of "merely declining" is therefore $\mathsf{await}_e\ c\ \omega.P$ with an environment that never offers dis_c. Its erasure offers dis_c, and the stable refusal of everything is a failure of the *composition* with an environment that refuses to discharge. The typed calculus thus moves declining out of the process and onto the environment, where the failures model is at home: the environment's refusal to discharge is a genuine CSP refusal, and it is not a disposition. A commitment that is declined in this sense stays live, and Theorem 23.1 names the hypothesis, responsiveness, whose failure it is.

11.5 Outcome

The second comparative claim is settled in its weak form. Recorded refusal is not represented by the failures model without instrumentation (Theorem 11.2), and CSP refusal sets and recorded refusals are independent notions, neither implying the other (Proposition 11.1). This is not a claim that the disposition calculus is more expressive than CSP. With the refusal events left visible, CSP carries the fact that a refusal happened, and with a rich enough alphabet it could carry the reason and the principal as well. The difference is in what each formalism treats as primary. The failures model is a theory of what a process can be observed not to do. The ledger is a record of what was decided. They classify different semantic objects, and a comparison that merged them would lose the distinction the book exists to keep.

Chapter 12

The Pi-Calculus: Restriction and Fresh Commitments

The π-calculus supplies what the disposition calculus most obviously resembles: fresh names, created by restriction and scoped lexically. Commitment identities behave like restricted names, and pop behaves like (νc). This chapter turns the resemblance into an encoding of the typed calculus into the π-calculus that satisfies all of the criteria of Chapter 4, and then shows exactly where the two calculi differ: not in expressive power, where, relative to the criteria used here, the π-calculus is strictly stronger, but in how they protect the integrity of a commitment's state.

One point about the target must be stated at the outset, since the encoding leans on it throughout. The lifecycle step δ, certificate checking, verifier execution, signature checking, and the observation functions $\mathsf{observe}_k$ are not implemented in the π-calculus. They are operations of the target's data theory, available in continuations as the conditionals and data expressions are. The encoding shows that the *process structure* of the disposition calculus is π-expressible given those operations; it does not show that the π-calculus computes them, and nothing here depends on whether it could.

12.1 The target

The target is the polyadic π-calculus with data values and conditionals, asynchronous output, and choice between guarded processes:

$$T ::= \mathbf{0} \mid x\langle \tilde{e} \rangle \mid \textstyle\sum_i G_i \mid T \mid T \mid (\nu x)T \mid !T, \qquad G ::= x(\tilde{y}).T \mid \tau.T,$$

where expressions e range over names and data, and conditionals and case analysis on data are available in continuations. Its reduction relation is the usual one: an output $x\langle \tilde{e} \rangle$ reacts with an input guard $x(\tilde{y}).T$ in a sum, discarding the other summands. The choice used here is *separate*: summands are guarded by inputs or by τ, never by outputs.

The encoding covers the *guarded* fragment of the typed calculus, in which every summand of a sum begins with a prefix. Every example in the book lies in it.

12.2 The encoding

A commitment c becomes a restricted name and a *cell*: an output $c\langle s, n\rangle$ holding its lifecycle state s and the number n of records written for it so far. The ledger becomes a multiset of outputs on a channel *log*, each tagged with its position in its commitment's projection, and observations become outputs on *obs*. For a configuration,

$$[\![\langle P, S, A, O, H\rangle]\!] = (\nu\ \mathrm{dom}(A))\Big(\prod_{c} c\langle A(c), |H|_c|\rangle \;\Big|\; \prod_{\rho\in H} log\langle \rho, \iota(\rho)\rangle \;\Big|\; \prod_{(c,k,o)\in O} obs\langle c, k, o\rangle \;\Big|\; [\![P]\!]\Big),$$

where $\iota(\rho)$ is the position of ρ in its projection, extended for a compensation record by the length its predecessor's projection had when the record was appended. Processes are encoded homomorphically on $\mathbf{0}$, $|$, $+$, (νc), and $!$, and a prefix that advances c reads its cell, writes it back, and logs:

$$[\![\mathsf{pop}\ q\ \mathsf{as}\ c.P]\!] = \tau.(\nu c)\big(c\langle \mathsf{open}(q), 1\rangle \mid log\langle \mathsf{Pop}(c, q), 0\rangle \mid [\![P]\!]\big),$$
$$[\![\mathsf{bind}_a\ c\ \mathsf{to}\ b.P]\!] = c(s, n).\big(c\langle \delta(s, \rho), n+1\rangle \mid log\langle \rho, n\rangle \mid [\![P]\!]\big) \quad \text{with } \rho = \mathsf{Bind}(a, c, b),$$

and transformation, refusal, and collapse likewise, the record computed from the state read and, for collapse, an *obs* output added. Verification reads the cell, runs the verifier on the value in it, and branches on the result into the successful or failed state and the corresponding continuation. Compensation reads the predecessor's cell and writes it back unchanged before creating the successor. An await first inputs a certificate on the environment channel ω and then updates the cell; the deadline form is a sum of two such environment inputs.

Three features are worth noticing. Freshness of commitment identities is restriction, with nothing added. The lifecycle checks of the operational rules have disappeared: in a well-typed configuration the prefix reading a cell always finds the state its rule requires (Lemma 14.7), so the encoding simply applies δ. And the ledger has become a multiset, with order kept only within each commitment.

Proposition 12.1 (The ledger as a trace)**.** *From the log outputs of $[\![C]\!]$, order two records when they have the same subject and smaller position, or when one is a compensation whose predecessor position covers the other. Every linearization of this order is trace equivalent to H.*

Proof. Within a subject, positions are the order of $H|_c$. A compensation record depends exactly on the records of its predecessor that precede it, and its recorded length identifies them. Records related by neither clause are independent (Definition 20.9). So the order is the dependence order of H, whose linearizations are its trace. □

The encoding recovers the history only up to trace equivalence, which by Proposition 20.10 is everything replay needs. It is the first comparison in the book in which the total order of the ledger is not merely unnecessary but absent.

12.3 The criteria

Theorem 12.2 (Encoding into the π-calculus)**.** *On the guarded fragment of the typed calculus, with success taken to be a log output of a designated kind, the encoding is*

(i) compositional, and homomorphic on every operator but the prefixes;

(ii) name invariant, with commitment names mapped to themselves;

(iii) operationally complete: each step $C \xrightarrow{\alpha} C'$ is matched by one reduction, or by two for a discharge or timeout, reaching a term structurally congruent to $[\![C']\!]$;

(iv) operationally sound: each reduction of $[\![C]\!]$ is the first of such a matching sequence for some step of C, provided the computations it starts terminate (Hypothesis 15.8);

(v) divergence reflecting; and

(vi) success sensitive.

Proof. (i) and (ii) are immediate from the definition, since the translation of a prefix mentions only its own names. (iii) By cases on the step: a pop is the τ of its encoding; a step on c is the reaction of the prefix with c's cell, after which the new cell, the new *log* output, and the continuation are the encoding of C'; a discharge or timeout is an environment input followed by the cell reaction. (iv) A reduction of $[\![C]\!]$ is a τ of an encoded pop, a reaction between an encoded prefix and a cell, or an environment input. In the second case either the prefix is a compensation reading its predecessor's cell, which by its typing premise holds a terminal state that it writes back unchanged, or the prefix occurrence holds the right for the cell's commitment, since in the typed calculus only the holder has an advancing prefix on it, and by canonical forms the cell holds the state that prefix requires; so the corresponding disposition step is enabled, and when its computation terminates the result is the encoding of its target. (v) No reduction is administrative beyond the one environment input in (iii), and there are no retry loops, because no prefix in a well-typed configuration waits for a state it does not already have. (vi) Success records correspond one to one. □

Two things in this proof depend on types, and they are the reason the encoding is stated for the typed calculus. Soundness needs the reading prefix to find the right state; in the untyped calculus a prefix may reach its cell early, and a faithful encoding would have to put the state back and retry, which introduces divergence and breaks (v). And the absence of a lock on cells is safe only because no two components ever act on one commitment.

Corollary 12.3 (Strict inclusion)**.** *The typed disposition calculus encodes into the π-calculus with data and separate choice, and the π-calculus does not encode into it compositionally and success sensitively.*

Proof. The first half is Theorem 12.2. For the second, CCS embeds into the π-calculus homomorphically, so an encoding of the π-calculus into the typed disposition calculus would compose to one of CCS, contradicting Theorem 10.5. □

Relative to these criteria and this fragment, the comparison therefore comes out against the disposition calculus. Every behaviour of the guarded typed fragment, with exclusive capabilities left out, has a compositional, name-invariant, operationally corresponding, divergence

reflecting, and success sensitive counterpart in the π-calculus with data and separate choice, built from restriction, cells, and a log, and using δ, the verifier, certificate checking, and the observation functions as operations of the data theory. Three qualifications keep the statement honest. It is relative to the criteria of Chapter 4, and a stricter criterion, such as full abstraction for a chosen equivalence, would need its own argument. It covers the guarded fragment only. And it does not cover delegation of exclusive capabilities, whose encoding would add a cell for each capability key (c, κ) holding its owner, read by every exercise and written by grants and delegations; that extension is routine in outline but has not been checked against the criteria here.

12.4 Where the difference lies: integrity

The encoding is correct in a closed world, where every commitment name remains restricted. Its correctness depends on that.

Proposition 12.4 (Scope versus authority)**.** *Let* $A(c) = \mathsf{open}(q)$.

(a) There is a π-process Att such that, if the name c is extruded to Att, then $[\![C]\!] \mid Att$ reduces to a term whose decoded ledger has no defined replay.

(b) For every disposition process Q whose principals hold no authority over c, typed or not, every execution of C with Q in parallel appends only authorized records for c and keeps its ledger consistent.

Proof. (a) Take $Att = c(s, n).(c\langle \mathsf{collapsed}(o), n+1\rangle \mid log\langle \mathsf{Collapse}(e, c, k, w, o), n\rangle)$. It reads the cell and logs a collapse of an open commitment, which δ rejects. (b) By Theorem 9.9 every record for c is attributed to a principal holding the required authority, which the principals of Q do not, so Q appends none; and every execution preserves consistency (Proposition 20.3), whoever runs it. □

The π-calculus protects a commitment's state by keeping its name secret. Whoever learns the name can write the cell. The disposition calculus protects it by authority and replay, and knowing a name confers nothing: this is the precise sense of the remark in Chapter 9 that restriction there scopes naming but not visibility. Scope extrusion, the π-calculus's defining feature, is exactly the operation under which its encoding of the disposition invariants fails.

The encoding can be hardened. Cells can check authority before writing, and the log can carry signatures that a decoder verifies. But each such check is a protocol written in the π-calculus rather than a guarantee the calculus supplies, and its correctness has to be proved for each protocol against each context. The disposition calculus builds those checks into its rules, and the safety theorems of Part V hold for every execution without a protocol proof.

12.5 Outcome

Strict inclusion in expressive power: the typed calculus is a sub-calculus of the π-calculus up to an encoding meeting every criterion (Corollary 12.3). The residue is not a construct but a guarantee. The π-calculus offers integrity through secrecy of names; the disposition calculus offers it through authority and an authenticated, replayable record, independently of who can name what (Proposition 12.4).

Chapter 13

Choice, Actors, and the Arbiter

Palamidessi showed that the π-calculus with mixed choice can elect a leader in a symmetric network, and that its separate-choice and asynchronous fragments cannot; from that she derived that no uniform encoding of the first into the second preserves a reasonable semantics. Choice, in that result, is where the power to break symmetry lives. This chapter asks where that power lives in the disposition calculus, and finds that it is not in choice at all.

13.1 Choice in the disposition calculus

The sum rule of Chapter 9 resolves $P + Q$ in favour of whichever summand acts first. No partner takes part: summands are guarded by acts of the process's own principals, not by communications. The π encoding of Chapter 12 accordingly uses only separate choice. By Palamidessi's result, then, whatever symmetry breaking the calculus can do does not come from its choice operator.

13.2 One race at three levels

The same race looks different at each level of the calculus, and it is worth following it through all three before stating anything about symmetry breaking. Let $A(c) = \mathsf{open}(q)$, and let principals a_1 and a_2 each hold refusal authority over c.

Level one: an untyped race on an unprotected map. Suppose the commitment map is shared by two replicas and updated without a linearization point. Each replica runs $\mathsf{refuse}_{a_i}\ c\ \mathsf{because}\ r_i.\mathbf{0}$. Both read $\mathsf{open}(q)$, both premises hold, and both append. The merged ledger

$$\mathsf{Pop}(c, q)\,;\mathsf{Refuse}(a_1, c, r_1)\,;\mathsf{Refuse}(a_2, c, r_2)$$

is not replay-defined: the second refusal meets a terminal state, and δ is undefined there. Ledger-level exclusivity survives, in the sense that replay rejects the history; operational exclusivity does not, since two components each proceeded as though it had refused. Nothing in the calculus has gone wrong. The execution simply violated Hypothesis 9.4, and the violation is detectable after the fact by any auditor who replays.

Level two: an atomic map elects a winner. With the hypothesis in force the race has a winner, and the network need not be asymmetric for it to have one.

Proposition 13.1 (Election by race)**.** *Let principals $a_1, \ldots, a_n$ each hold refusal authority over c, with $A(c) = \mathsf{open}(q)$. Under Hypothesis 9.4, every maximal execution of the untyped process*

$$\prod_{i=1}^{n} \mathsf{refuse}_{a_i}\ c\ \mathsf{because}\ r.\, W_i$$

appends exactly one refusal of c, attributed to one principal a_j, after which W_j runs and no other W_i ever starts.

Proof. Each component's prefix is enabled while c is live. The first to fire makes c terminal, which disables every other prefix permanently by lifecycle monotonicity. Operational terminal exclusivity rests on the atomicity of the commitment map. □

As stated, the losers do not learn that they lost; they are simply blocked. They can be made to learn it. Compensation's premise is that its predecessor is terminal, and a compensation prefix is therefore a test of terminality that becomes enabled exactly when the race is over.

Proposition 13.2 (Losers can detect the loss)**.** *Let each a_i also hold compensation authority on c, and let*

$$R_i = \mathsf{refuse}_{a_i}\ c\ \mathsf{because}\ r.\, W_i\ +\ \mathsf{pop}_{a_i}\ q_i\ \mathsf{as}\ c_i\ \mathsf{after}\ c.\, L_i.$$

Under Hypothesis 9.4, in every execution of $\prod_i R_i$ the first act on c is a refusal by some a_j, after which W_j runs, and every other component can only take its second summand; under weak fairness each of them does, and runs L_i.

Proof. While c is live no compensation premise holds, so the first act is a refusal. After it, every other refusal prefix is disabled forever and every other compensation prefix is enabled forever, because terminal states are final. The names c_i are fresh by the generative binder, so the compensations are independent of one another and of W_j. □

So the atomic map, used this way, is a test-and-set object: one caller receives "won" and every other caller eventually receives "lost". The successors c_i are the competing compensations of Appendix E, admitted by replay and a question for policy.

Level three: the typed calculus rejects the race.

Corollary 13.3 (The typed calculus cannot race)**.** *No well-typed program contains the race of Proposition 13.1 or Proposition 13.2, and no well-typed program elects a leader among its own parallel components.*

Proof. Typing $R_1 \mid R_2$ splits the context, $\Delta = \Delta_1 \uplus \Delta_2$, and the refusal prefix of each R_i needs the linear right $\mathrm{Commit}\langle c, A, \mathsf{Open}\rangle$ in its own part. A linear right lies in at most one part of a split, so one of the two premises fails and no derivation exists (Lemma 14.1). More generally, by Lemma 10.3 no component's behaviour depends on another's, so no component can learn that it lost. □

13.3 What the race does and does not give

The comparison with Herlihy's hierarchy needs care, because the calculus is not the shared-memory model in which that hierarchy is stated. In that model test-and-set has consensus number two: together with read/write registers it solves wait-free consensus among two processes and not among three. The atomic commitment map provides the test-and-set part. It does not provide the registers. A disposition process cannot read the commitment map; it can only attempt acts whose premises consult it, and learn, by which prefix fires, one bit about the state. In particular a loser in Proposition 13.2 learns that c is terminal, and, since terminality here can only come from a refusal, that someone else refused. It does not learn who, or for what reason, since no premise of any rule exposes the refusing principal or the reason to a process.

Three consequences follow, stated with their scope. Leader election among any number of components is solvable by this mechanism, in the weak form that each component learns whether it is the leader. Consensus on proposed values is not solvable by this mechanism alone, even between two components, because the loser has no way to read the winner's proposal; a calculus that let processes read replay state, or that carried values over channels as the π-calculus does, would recover the usual construction. And the claim "the commitment map has consensus number two" should not be made of the calculus as it stands, since consensus number is defined relative to an object together with registers, and the calculus has no readable registers. What can be said is that the atomic map is a test-and-set object and that its symmetry-breaking power, in the form leader election needs, is exercised by the untyped race.

In a single-sequencer implementation this object is the arbiter. So the arbiter, not the choice operator, is what breaks symmetry; the kernel's single-sequencer assumption is doing real work in the untyped calculus.

The typed discipline confines the arbiter's power to a place where it is never exercised. That is the same fact, seen from the other side, as the coordination-free convergence of Theorem 22.3: programs that never race need no arbiter.

13.4 Recording the road not taken

When $P + Q$ is resolved in favour of P, nothing records that Q was available. A variant rule could append, on the annotation channel, the first prefixes of the discarded summands. Since annotations take part in no causal reduction, the variant changes nothing that replay, the path lemma, or any safety theorem consults, and none of the criteria of Chapter 4, which do not observe annotations, can tell the variants apart. The variant is more informative, telling an auditor what alternatives were open, and not more expressive.

13.5 Actors and the order of arrival

In the actor model a mailbox receives messages in an order no one chooses. When a disposition ledger records an actor-like system, an arbiter fixes that order. The Causal Shuffle theorem says which parts of the fixed order carry meaning.

Proposition 13.4 (Order artifacts)**.** *A lens on histories that reports the relative order of two independent records is not invariant under trace equivalence, and so is not a function of the*

execution. Every lens that factors through replay is.

Proof. Exchanging the two records gives a trace-equivalent history on which the first lens differs. The second part is Proposition 20.10. □

"Which message arrived first" is therefore meaningful only for messages about the same commitment, where the order is a dependency the ledger must keep. For messages about different commitments it is an artifact of the serialization the arbiter chose, and a system that makes decisions on it is deciding on the arbiter's convenience rather than on anything that happened.

13.6 Outcome

The disposition calculus's choice is separate choice, and it breaks no symmetry. The untyped calculus breaks symmetry anyway, through its atomic commitment map, which is the arbiter; the typed calculus never uses that power. The comparison with Palamidessi's hierarchy is therefore not a statement about choice. It is a statement about where a calculus keeps its shared, atomically updated state, and the typed discipline's answer is that it keeps none that two components can contend for.

Part IV

Linear and Session Types

Chapter 14

The Type Algebra

The untyped calculus says what happens when a step is enabled. It says nothing about programs that attempt steps that can never be enabled: a collapse of an open commitment, a second binding of the same commitment from two parallel components, a refusal by a principal without authority. Such programs simply stop, with nothing recorded. The type algebra excludes them. It does so by making the right to advance a commitment a linear resource, indexed by the commitment's identity and lifecycle state, and by stating exactly how those resources correspond to the runtime commitment map.

14.1 Types, rights, and facts

The value types are

$$A, B ::= \mathbf{1} \mid A \times B \mid A + B \mid A \to B \mid A \otimes B \mid A \multimap B \mid !A.$$

On top of them the disposition layer uses three kinds of assumption.

- *Rights* $\mathsf{Commit}\langle c, A, s\rangle$, for a live state s: one of Open, Bound, $\mathsf{Transformed}$, $\mathsf{Verified}$, and $\mathsf{VerificationFailed}$. A right is the permission to advance commitment c from state s. Rights are linear: they may be neither duplicated nor discarded.
- *Tokens* $\mathsf{Refusal}\langle c, A, s\rangle$ and $\mathsf{Observed}\langle c, B, k\rangle$, the results of the two terminal steps. The index s of a refusal token records the stage at which refusal occurred. Tokens are affine: they may be discarded but not duplicated.
- *Facts*: the authority policy $\mathsf{may}(a, \kappa, x)$, failure evidence $\mathsf{FailureEvidence}\langle c, B\rangle$ recording that verification of c returned a certified negative result, and terminal facts $\mathsf{Terminal}\langle c, t\rangle$ with $t \in \{\mathsf{Refused}, \mathsf{Collapsed}\}$. Facts are persistent. A terminal fact is issued by the terminal step of c and may be consulted any number of times afterward, which is sound because terminality is monotone (lifecycle monotonicity).

The linear context holds the right to advance a commitment, never the historical fact that the commitment exists. The history and the commitment map keep c forever (Theorem 9.5); the context holds at most one right to move it.

The typing judgement for processes is

$$\Sigma; \Gamma; \Delta \vdash P,$$

where Σ is a set of names (commitment identities and obligations), Γ contains value typings and facts, and Δ contains rights and tokens. The protocol Θ offered by P is, in this edition, determined by Δ: each right $\mathsf{Commit}\langle c, A, s\rangle$ offers the row of the refusal completion table for s, read as a protocol (Section 15.4). The general judgement $\Sigma; \Gamma; \Delta \vdash P \triangleright \Theta$ with independently specified protocols is needed only for interaction among several principals over shared channels, which this calculus does not yet have.

14.2 Operator types

Read as functions, the disposition operators have these types.

$$
\begin{aligned}
\mathsf{bind} &: \mathsf{Commit}\langle c, A, \mathsf{Open}\rangle \otimes A \multimap \mathsf{Commit}\langle c, A, \mathsf{Bound}\rangle,\\
\mathsf{transform} &: \mathsf{Commit}\langle c, A, \mathsf{Bound}\rangle \otimes (A \to B) \multimap \mathsf{Commit}\langle c, B, \mathsf{Transformed}\rangle,\\
\mathsf{verify} &: \mathsf{Commit}\langle c, B, \mathsf{Transformed}\rangle \otimes \mathsf{Verifier}\langle c, B\rangle\\
&\qquad \multimap \big(\exists w.\ \mathsf{Commit}\langle c, B, \mathsf{Verified}(w)\rangle \otimes \mathsf{Evidence}\langle c, B, w\rangle\big)\\
&\qquad \oplus\ \big(\exists \varphi.\ \mathsf{Commit}\langle c, B, \mathsf{VerificationFailed}(\varphi)\rangle \otimes\, !\mathsf{FailureEvidence}\langle c, B, \varphi\rangle\big),\\
\mathsf{refuse}_s &: \mathsf{Commit}\langle c, A, s\rangle \otimes \mathsf{Reason}\\
&\qquad \multimap \mathsf{Refusal}\langle c, A, s\rangle \otimes\, !\mathsf{Terminal}\langle c, \mathsf{Refused}\rangle,\\
\mathsf{collapse}_k &: \mathsf{Commit}\langle c, B, \mathsf{Verified}(w)\rangle\\
&\qquad \multimap \mathsf{Observed}\langle c, B, k\rangle \otimes\, !\mathsf{Terminal}\langle c, \mathsf{Collapsed}\rangle.
\end{aligned}
$$

Four features deserve comment.

Pop is generative, not an ordinary existential. Its external type may be written $A \to \exists c.\, \mathsf{Commit}\langle c, A, \mathsf{Open}\rangle$, but internally it is a name-generation rule analogous to restriction: its typing premise extends Σ with a name not already in it, and the continuation receives the open right for that name (rule T-Pop below). An ordinary existential could be eliminated and its witness compared with other names; a generated name is fresh by construction, and that is what freshness of commitment identities rests on.

The success branch of verification carries its witness. The verified state is indexed by the witness w, and the evidence $\mathsf{Evidence}\langle c, B, w\rangle$ is indexed by the commitment. Evidence for one commitment is never evidence for another: the index is part of the type, and under Hypothesis 2.10 it is part of the predicate too. In process typing the witness does not need a name, because collapse reads it from the runtime state; what the typing guarantees is that such a witness exists there (Lemma 14.7).

Both verification branches keep the right. The positive branch produces a verified right with its evidence; the negative branch produces a right in VerificationFailed with persistent failure evidence. Neither produces a refusal token or a terminal fact, because neither is a disposition. A verifier cannot end a commitment, and the type says so.

Terminal steps issue persistent facts. Refusal and collapse consume the right and produce two things: an affine token recording the result, and a persistent fact $\mathsf{Terminal}\langle c, t\rangle$. The fact is what compensation consults. The earlier worry, that compensation needed a certificate of terminality that no rule issued, is resolved by making the terminal rules issue it.

14.3 Context algebra

A *linear context* Δ is a finite multiset of rights and tokens. It is *well formed* when it contains at most one right for each commitment, at most one token for each commitment, and never both a right and a token for the same commitment. Write $\Delta_1 \uplus \Delta_2$ for multiset union, defined only when the result is well formed.

Lemma 14.1 (Ownership)**.** *If $\Delta_1 \uplus \Delta_2$ is defined and Δ_1 contains a right for c, then Δ_2 contains neither a right nor a token for c.*

Proof. Otherwise the union would contain two assumptions for c of kinds that well-formedness forbids together. □

For an action α with subject c, the *context step* $\Delta \xrightarrow{\alpha} \Delta'$ is defined by: a pop or compensation adds $\mathsf{Commit}\langle c, A, \mathsf{Open}\rangle$; binding, transformation, and successful verification replace the right for c by the right for its successor state; either kind of refusal replaces the right by the token $\mathsf{Refusal}\langle c, A, s\rangle$; collapse replaces the right by $\mathsf{Observed}\langle c, B, k\rangle$; a discharge leaves Δ unchanged.

Lemma 14.2 (Recombination)**.** *If $\Delta_1 \uplus \Delta_2$ is defined, $\Delta_1 \xrightarrow{\alpha} \Delta_1'$, and either α introduces a name not mentioned in Δ_2 or Δ_1 contains the right for the subject of α, then $\Delta_1' \uplus \Delta_2$ is defined and equals $(\Delta_1 \uplus \Delta_2)'$, the context step applied to the union.*

Proof. The context step changes only the entry for the subject of α. In the first case that entry is new to both sides. In the second, by Lemma 14.1, Δ_2 has no entry for it. Either way the changed entry lives in Δ_1' alone and well-formedness of the union is preserved. □

The unrestricted context Γ admits weakening, contraction, and exchange. The linear context admits exchange only, together with weakening by tokens.

14.4 Typing the value layer

Values are typed by $\Gamma; \Delta \vdash e : A$. The rules for the multiplicative core are

$$\frac{}{\Gamma, x{:}A; \varnothing \vdash x : A} \qquad \frac{}{\Gamma; x{:}A \vdash x : A} \qquad \frac{\Gamma; \Delta, x{:}A \vdash e : B}{\Gamma; \Delta \vdash \lambda x{:}A.e : A \multimap B} \qquad \frac{\Gamma, x{:}A; \varnothing \vdash e : B}{\Gamma; \varnothing \vdash \lambda x{:}A.e : A \to B}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : A \multimap B \qquad \Gamma; \Delta_2 \vdash e_2 : A}{\Gamma; \Delta_1 \uplus \Delta_2 \vdash e_1\, e_2 : B} \qquad \frac{\Gamma; \Delta \vdash e_1 : A \to B \qquad \Gamma; \varnothing \vdash e_2 : A}{\Gamma; \Delta \vdash e_1\, e_2 : B}$$

with products, sums, projections, injections, and case analysis typed in the unrestricted fragment in the standard way. Transformations and verifiers used by processes are closed unrestricted terms.

Lemma 14.3 (Unrestricted substitution)**.** *If $\Gamma, x{:}A; \Delta \vdash e : B$ and $\Gamma; \varnothing \vdash v : A$, then $\Gamma; \Delta \vdash e[v/x] : B$.*

Proof. Induction on the derivation of e. At an unrestricted variable x the result is the typing of v, weakened in Γ. At a rule that splits Δ, the variable x lives in Γ and so is available on both sides, and the induction hypothesis applies to each. At a binder, rename the bound variable apart from x and the free variables of v first. □

Lemma 14.4 (Linear substitution)**.** *If* $\Gamma; \Delta_1, x{:}A \vdash e : B$ *and* $\Gamma; \Delta_2 \vdash v : A$*, and* $\Delta_1 \uplus \Delta_2$ *is defined, then* $\Gamma; \Delta_1 \uplus \Delta_2 \vdash e[v/x] : B$*.*

Proof. Induction on the derivation of e. At a linear variable, $e = x$ and $\Delta_1 = \varnothing$, so the result is the typing of v. The linear variable x cannot occur at an unrestricted variable rule or in the body of an unrestricted abstraction, since both require an empty linear context. At linear application, $\Delta_1, x{:}A$ is split between the two premises and x lands in exactly one of them; apply the induction hypothesis to that premise and recombine, the resources of v joining the side that held x. At unrestricted application the argument has an empty linear context, so x is in the function premise. At linear abstraction, rename apart and apply the induction hypothesis. In no case are the resources of v duplicated, because x occurs in exactly one branch of every split. □

Evaluation $e \Downarrow v$ preserves types for closed terms, by induction on the evaluation derivation using the two substitution lemmas; closed values of function type are abstractions, of product type pairs, and of sum type injections. These canonical-forms facts are standard [28] and are used without further proof.

14.5 Typing processes

The process rules are syntax directed. Write $\Gamma \vdash e : A$ for $\Gamma; \varnothing \vdash e : A$.

$$\textsc{T-Nil}\ \frac{\Delta \text{ contains only tokens}}{\Sigma; \Gamma; \Delta \vdash \mathbf{0}} \qquad \textsc{T-Pop}\ \frac{c \notin \Sigma \quad \Gamma \vdash q : A \quad \Sigma, c; \Gamma; \Delta, \mathrm{Commit}\langle c, A, \mathsf{Open}\rangle \vdash P}{\Sigma; \Gamma; \Delta \vdash \mathsf{pop}\ q\ \mathsf{as}\ c.P}$$

$$\textsc{T-Comp}\ \frac{\begin{array}{c} c \notin \Sigma \quad \mathsf{Terminal}\langle d, t\rangle \in \Gamma \quad \mathsf{may}(a, \mathsf{compensate}, d) \in \Gamma \quad \Gamma \vdash q : A \\ \Sigma, c; \Gamma; \Delta, \mathrm{Commit}\langle c, A, \mathsf{Open}\rangle \vdash P \end{array}}{\Sigma; \Gamma; \Delta \vdash \mathsf{pop}_a\ q\ \mathsf{as}\ c\ \mathsf{after}\ d.P}$$

$$\textsc{T-Bind}\ \frac{\mathsf{may}(a, \mathsf{bind}, c) \in \Gamma \quad \Gamma \vdash b : A \quad \Sigma; \Gamma; \Delta, \mathrm{Commit}\langle c, A, \mathsf{Bound}\rangle \vdash P}{\Sigma; \Gamma; \Delta, \mathrm{Commit}\langle c, A, \mathsf{Open}\rangle \vdash \mathsf{bind}_a\ c\ \mathsf{to}\ b.P}$$

$$\textsc{T-Trans}\ \frac{\Gamma \vdash f : A \to B \quad \Sigma; \Gamma; \Delta, \mathrm{Commit}\langle c, B, \mathsf{Transformed}\rangle \vdash P}{\Sigma; \Gamma; \Delta, \mathrm{Commit}\langle c, A, \mathsf{Bound}\rangle \vdash \mathsf{transform}\ c\ \mathsf{with}\ f.P}$$

$$\textsc{T-Ver}\ \frac{\begin{array}{c} \mathsf{may}(a, \mathsf{verify}, c) \in \Gamma \qquad \Gamma \vdash g : \mathsf{Verifier}\langle c, B\rangle \\ \Sigma; \Gamma; \Delta, \mathrm{Commit}\langle c, B, \mathsf{Verified}\rangle \vdash P \\ \Sigma; \Gamma, \mathsf{FailureEvidence}\langle c, B\rangle; \Delta, \mathrm{Commit}\langle c, B, \mathsf{VerificationFailed}\rangle \vdash Q \end{array}}{\Sigma; \Gamma; \Delta, \mathrm{Commit}\langle c, B, \mathsf{Transformed}\rangle \vdash \mathsf{verify}_a\ c\ \mathsf{using}\ g\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

$$\textsc{T-Ref}\ \frac{\begin{array}{c} \mathsf{live}(s) \quad \mathsf{may}(a, \mathsf{refuse}, c) \in \Gamma \quad \Gamma \vdash r : \mathsf{Reason} \\ \Sigma; \Gamma, \mathsf{Terminal}\langle c, \mathsf{Refused}\rangle; \Delta, \mathsf{Refusal}\langle c, A, s\rangle \vdash P \end{array}}{\Sigma; \Gamma; \Delta, \mathrm{Commit}\langle c, A, s\rangle \vdash \mathsf{refuse}_a\ c\ \mathsf{because}\ r.P}$$

$$\textsc{T-Col}\ \frac{\begin{array}{c} \mathsf{may}(a, \mathsf{collapse}, c) \in \Gamma \quad k : \mathsf{Rule}(B) \\ \Sigma; \Gamma, \mathsf{Terminal}\langle c, \mathsf{Collapsed}\rangle; \Delta, \mathsf{Observed}\langle c, B, k\rangle \vdash P \end{array}}{\Sigma; \Gamma; \Delta, \mathrm{Commit}\langle c, B, \mathsf{Verified}\rangle \vdash \mathsf{collapse}_{a,k}\ c.P}$$

$$\textsc{T-Await}\ \frac{\omega \in \Sigma \quad \mathsf{may}(a, \mathsf{discharge}, \omega) \in \Gamma \quad \Sigma; \Gamma; \Delta, \mathrm{Commit}\langle c, A, s\rangle \vdash P}{\Sigma; \Gamma; \Delta, \mathrm{Commit}\langle c, A, s\rangle \vdash \mathsf{await}_a\ c\ \omega.P}$$

$$\textsc{T-Deadline}\ \frac{\begin{array}{c} \omega \in \Sigma \quad \mathsf{may}(a, \mathsf{discharge}, \omega),\ \mathsf{may}(b, \mathsf{clock}, \omega) \in \Gamma \\ \Sigma; \Gamma; \Delta, \mathrm{Commit}\langle c, A, s\rangle \vdash P \qquad \Sigma; \Gamma; \Delta, \mathrm{Commit}\langle c, A, s\rangle \vdash Q \end{array}}{\Sigma; \Gamma; \Delta, \mathrm{Commit}\langle c, A, s\rangle \vdash \mathsf{await}_a\ c\ \omega.P\ \mathsf{within}_b\ t\ \mathsf{else}\ Q}$$

$$\textsc{T-Par}\ \frac{\Sigma; \Gamma; \Delta_1 \vdash P \quad \Sigma; \Gamma; \Delta_2 \vdash Q}{\Sigma; \Gamma; \Delta_1 \uplus \Delta_2 \vdash P \mid Q} \qquad \textsc{T-Sum}\ \frac{\Sigma; \Gamma; \Delta \vdash P \quad \Sigma; \Gamma; \Delta \vdash Q}{\Sigma; \Gamma; \Delta \vdash P + Q}$$

$$\textsc{T-Res}\ \frac{c \notin \Sigma \quad \Sigma, c; \Gamma; \Delta \vdash P}{\Sigma; \Gamma; \Delta \vdash (\nu c)P} \qquad \textsc{T-Rep}\ \frac{\Sigma; \Gamma; \varnothing \vdash P}{\Sigma; \Gamma; \varnothing \vdash\ !P}$$

Here $\Gamma \vdash g : \mathsf{Verifier}\langle c, B\rangle$ says g is a verifier whose witnesses and certificates are issued for c, and $k : \mathsf{Rule}(B)$ says $\mathsf{observe}_k$ is total on B.

Several design decisions are visible in the rules. T-Nil forbids ending a process while it holds a right: a well-typed program cannot silently abandon a live commitment. T-Sum types both summands against the same context, so a choice is between alternative uses of the same right. T-Rep requires an empty linear context, so no right is ever replicated. The terminal rules add the terminal fact to Γ for the continuation, and T-Comp consults it. Every attributed prefix requires its authority fact, so an unauthorized attempt cannot be written in a well-typed program at all.

14.6 Configurations and the rights–state correspondence

A typing of the process says nothing, by itself, about the runtime. The bridge between them is the following invariant. Without it the commitment map and the linear context could each be internally consistent while disagreeing with each other.

Definition 14.5 (Rights agree)**.** $\mathsf{RightsAgree}(A, \Delta)$ *holds when, for every commitment* c*:*

(i) if $A(c)$ *is live with state kind* s*, then* Δ *contains exactly one right for* c*, namely* $\mathrm{Commit}\langle c, A_c, s\rangle$*, and the values held in* $A(c)$ *have the types that right indexes;*

(ii) if $A(c)$ *is terminal or* $c \notin \mathrm{dom}(A)$*, then* Δ *contains no right for* c*;*

(iii) a token for c *in* Δ *occurs only if* $A(c)$ *is terminal, and a refusal token only if* $A(c)$ *is refused, an observation token only if it is collapsed.*

Definition 14.6 (Well-typed configuration)**.** $\Sigma; \Gamma; \Delta \vdash \langle P, S, A, O, H\rangle\ \mathsf{ok}$ *when*

(C1) $\Sigma; \Gamma; \Delta \vdash P$*;*

(C2) the configuration is consistent: $\mathrm{replay}(H) = \langle A, O\rangle$*;*

(C3) $\mathsf{RightsAgree}(A, \Delta)$*;*

(C4) every fact in Γ *is true: the authority facts are the policy, and* $\mathsf{Terminal}\langle c, t\rangle \in \Gamma$ *only if* $A(c)$ *is terminal of kind* t*;*

(C5) $\mathrm{dom}(A) \subseteq \Sigma$*, and the names occurring free in* P *are in* Σ*.*

Lemma 14.7 (Canonical forms for rights)**.** *In a well-typed configuration, if* Δ *contains* $\mathrm{Commit}\langle c, B, \mathsf{Verified}\rangle$*, then* $A(c) = \mathsf{verified}(v, w)$ *with* v *of type* B *and* $\mathrm{valid}(w, c, v)$*. If* Δ *contains* $\mathrm{Commit}\langle c, B, \mathsf{VerificationFailed}\rangle$*, then* $A(c) = \mathsf{verificationFailed}(v, \varphi)$ *with* $\mathsf{checkCert}(\varphi, c, v, \mathsf{fail})$*. If* Δ *contains* $\mathrm{Commit}\langle c, A, s\rangle$ *for any live* s*, then* $A(c)$ *is in state* s *and* $H|_c$ *is a* δ*-path from an opening record to* $A(c)$*.*

Proof. By (C3) the state of c is s with correctly typed contents. By (C2) and the definition of replay, $A(c) = \delta^*(H|_c)$, which is a path; for the verified case the verification clause of δ requires $\mathrm{valid}(w, c, v)$, for the failed case the failure clause requires the certificate check, and by stability both still hold. □

Canonical forms is where the evidence discipline becomes a static guarantee: possession of a verified right is enough to know that a correctly indexed witness exists in the state, without the program ever having named it.

14.7 The proof target

The metatheory of the next chapter establishes preservation, progress, and protocol fidelity. Their synthesis is the following statement, proved in Chapter 26.

Theorem 14.8 (Typed disposition safety, target form)**.** *Suppose* $\Sigma; \Gamma; \Delta \vdash C_0$ ok*. For every finite execution* $C_0 \stackrel{\tau}{\Longrightarrow} C_n$*, the configuration* C_n *is well typed; every live commitment has exactly one right to advance it and every terminal commitment has none; every attributed transition was authorized; every witness is indexed by the commitment and value it verifies; every collapse has a complete verified ledger path; every refusal remains in the history; no commitment has two terminal dispositions; and no well-typed program contains a competing continuation for a commitment whose right it does not hold.*

Chapter 15

Preservation, Progress, and Protocol Fidelity

This chapter proves that typing is preserved by execution, that a well-typed configuration is never silently stuck, and that every execution follows the lifecycle protocol of each commitment it touches. It closes with the negative derivations: the programs the type system rejects, and why.

15.1 Structural lemmas

Lemma 15.1 (Inversion)**.** *Each typing rule for a prefix is the only rule whose conclusion has that prefix at the root. Hence from a derivation of* $\Sigma;\Gamma;\Delta \vdash \pi.P$ *the premises of the corresponding rule can be read off; in particular* Δ *contains the right the rule consumes.*

Proof. The rules are syntax directed; only T-Nil, T-Par, T-Sum, T-Res, and T-Rep apply to non-prefix forms, and each applies to exactly one form. □

Lemma 15.2 (Weakening and strengthening)**.** *Typing is preserved by adding true facts to* Γ*, by adding tokens to* Δ*, and by adding to or removing from* Σ *names that occur neither free in* P *nor in* Δ*.*

Proof. Induction on the derivation. Facts are only ever consulted, tokens are absorbed by T-Nil, and names outside the free names are never inspected except by freshness premises, which remain satisfiable by choosing binders apart. □

Lemma 15.3 (Equivariance of typing)**.** *For every permutation* ρ *of names,* $\Sigma;\Gamma;\Delta \vdash P$ *implies* $\rho\Sigma;\rho\Gamma;\rho\Delta \vdash \rho P$*.*

Proof. Induction on the derivation; every premise that mentions names compares them only for equality or membership, which permutations preserve. □

Lemma 15.4 (Congruence)**.** *If* $P \equiv Q$ *then* $\Sigma;\Gamma;\Delta \vdash P$ *iff* $\Sigma;\Gamma;\Delta \vdash Q$*.*

Proof. By cases on the axioms, each in both directions. The laws for $\mid$ are commutativity and associativity of $\uplus$ together with T-Nil for the unit. Commutativity of $+$ exchanges the two identical-context premises of T-Sum. For $!P \equiv P \mid !P$, T-Rep gives an empty context, and

$\varnothing \uplus \varnothing = \varnothing$. The restriction laws use Lemma 15.2 for names not free, and α-conversion uses Lemma 15.3. □

15.2 Preservation

Theorem 15.5 (Preservation)**.** *If* $\Sigma; \Gamma; \Delta \vdash C\ \mathsf{ok}$ *and* $C \xrightarrow{\alpha} C'$*, then there are* $\Sigma' \supseteq \Sigma$ *and* $\Gamma' \supseteq \Gamma$ *such that* $\Sigma'; \Gamma'; \Delta' \vdash C'\ \mathsf{ok}$*, where* $\Delta \xrightarrow{\alpha} \Delta'$ *is the context step.*

Proof. By induction on the derivation of the transition, with the process typing brought into matching shape by Lemma 15.4.

Primitive rules. By Lemma 15.1 the prefix's typing premises are available.

- *Pop.* The premise of T-Pop types P under Σ, c with the open right; the operational rule chose c outside $\mathrm{dom}(A)$, and by Lemma 15.3 we may take the binder to be that name. (C2) holds since $\delta(\bot, \mathsf{Pop}(c, q)) = \mathsf{open}(q)$ and $c \notin \mathrm{dom}(A)$ means $H|_c$ was empty. (C3) holds with the new right, and $q : A$ by the premise.

- *Compensating pop.* As for pop, with the terminal fact for d in Γ matching the operational premise by (C4).

- *Bind, transform, successful verification.* Inversion gives the right for the prior state in Δ and types the continuation with the right for the next state. By (C3) the runtime state is the prior one, the operational rule moves it to the next, and δ agrees, so (C2) and (C3) hold. For transformation, $f\, b \Downarrow v$ with $f : A \to B$ and $b : A$ gives $v : B$ by type preservation of evaluation. For verification, v keeps its type and the witness is valid by Hypothesis 1.5.

- *Failed verification.* Inversion of T-Ver types Q with the right for c in $\mathsf{VerificationFailed}$ and the persistent fact $\mathsf{FailureEvidence}\langle c, B\rangle$. The operational rule moves c to $\mathsf{verificationFailed}(v, \varphi)$ and δ agrees, so (C2) and (C3) hold, and the fact is true in C', so (C4) holds.

- *Refusal.* The right is replaced by the refusal token, the continuation is typed with the terminal fact added to Γ, and the fact is true in C', so (C4) holds with $\Gamma' = \Gamma, \mathsf{Terminal}\langle c, \mathsf{Refused}\rangle$.

- *Collapse.* As for refusal, with the observation token and the collapsed fact; $o = \mathsf{observe}_k(v)$ is defined since k is total on B.

- *Discharge and timeout.* The awaiting process holds the right for c (T-Await, T-Deadline), so c is live by (C3), and the continuation taken, P or Q, keeps the right. The corresponding clause of δ leaves the state unchanged, so (C2) and (C3) hold with $\Delta' = \Delta$.

Contextual rules. For $P \mid Q$, T-Par splits $\Delta = \Delta_1 \uplus \Delta_2$. The step is taken by P, and by the induction hypothesis P' is typed under Δ_1'. The prefix that fired either introduced a name fresh for Σ, hence not mentioned in Δ_2, or consumed a right in Δ_1; Lemma 14.2 gives $\Delta_1' \uplus \Delta_2 = \Delta'$, and T-Par retypes $P' \mid Q$ after weakening Q's typing with the new facts and names (Lemma 15.2). For $P + Q$, the premise of T-Sum types P under the same Δ, and the induction hypothesis applies. For restriction, the induction hypothesis applies under Σ, c. For the congruence rule, Lemma 15.4.

The conditions on the other components of the configuration are carried over unchanged or updated as described in each case. □

Corollary 15.6 (Rights are neither duplicated nor lost)**.** *In every configuration reachable from a well-typed one, each live commitment has exactly one right in the linear context and each terminal commitment has none.*

Proof. (C3) is preserved by Theorem 15.5. □

Corollary 15.7 (Typed path lemma)**.** *In a well-typed configuration, if* $\mathrm{Commit}\langle c, A, s\rangle \in \Delta$ *then* $H|_c$ *is a* δ*-path from an opening record to a state of kind* s*. It is unique in the sense that every linearization of the history's dependence order has the same projection onto* c*.*

Proof. The first sentence is Lemma 14.7. For the second, records of the same subject are pairwise dependent, so every linearization keeps them in the same relative order. □

15.3 Progress

Progress in an ordinary calculus says a well-typed term is a value or can step. Here a third possibility is legitimate: waiting for the environment. The danger is that "waiting" becomes a place for errors to hide. It cannot here, because waiting is a positive syntactic form, the await prefix, which names the obligation and the principal expected to discharge it.

A process is *inert* if it is structurally congruent to a parallel composition of **0** and replications of processes with no prefix. A configuration is *waiting on* ω if every top-level prefix of its process is an await prefix, with or without a deadline, and one of them awaits ω. The *top-level prefixes* of P are those not beneath another prefix; a summand's prefixes count, and a replicated body's prefixes count.

Hypothesis 15.8 (Computational totality at C)**.** *For every top-level prefix* transform c with f *of* C *with* $A(c) = \mathsf{bound}(q, b)$*, evaluation of* $f\,b$ *terminates; and for every top-level prefix* $\mathsf{verify}_a\ c$ using g *with* $A(c) = \mathsf{transformed}(b, v)$*, the verifier* $g(c, v)$ *returns* ok *or* fail*.*

The hypothesis is about the particular computations a configuration is about to perform, not about the lambda layer as a whole, which may be Turing complete. For the fragment without fixed points it holds automatically by strong normalization of the simply typed calculus [27].

Theorem 15.9 (Progress)**.** *Let* $\Sigma; \Gamma; \Delta \vdash C$ ok *with Hypothesis 15.8 holding at* C*. Then exactly one of the following holds:*

(i) C *can take a transition other than a discharge;*

(ii) C *is waiting on some obligation;*

(iii) the process of C *is inert, and every commitment in* $\mathrm{dom}(A)$ *is terminal.*

Proof. If some top-level prefix is not an await, we show it is enabled, so (i) holds. By Lemma 15.1 its typing premises hold, and by (C3) and (C4) the runtime agrees with them.

- A pop is always enabled: names are unbounded, so some c lies outside $\mathrm{dom}(A)$.

- A compensating pop has $\mathsf{Terminal}\langle d, t\rangle$ in its Γ, true by (C4), and its authority fact; a fresh name exists.

- A bind, transform, verify, refuse, or collapse prefix on c holds the right for the state it requires (inversion), and by (C3) $A(c)$ is in that state. Authority is a premise of the typing. A transform or verify terminates by Hypothesis 15.8. A collapse finds a valid witness in the state by Lemma 14.7 and a defined observation since k is total on the value's type.

- A prefix under a sum, restriction, or replication fires through the corresponding contextual rule; for replication, through $!R \equiv R \mid !R$.

If every top-level prefix is an await and there is at least one, (ii) holds. If there is none, the process is inert; its typing derivation ends in instances of T-Nil, so Δ contains only tokens, and by (C3) no commitment is live.

The three cases are exclusive by definition of the second and third, and since a configuration with an enabled non-await prefix is neither waiting nor inert. □

The third case is the formal statement that a well-typed program cannot silently abandon a commitment: if nothing can happen and nothing is being waited for, every commitment has been disposed of. A process attempting to collapse an unverified commitment is not waiting and not inert; it is untypable.

15.4 Protocol fidelity

Read the refusal completion table as a protocol: for each live state s, the protocol offers the actions in $\mathsf{Next}(s)$, and each action leads to the protocol of its successor state. For a single commitment this is the session type

$$\mathcal{S} = \mathsf{open};\big(\mathsf{refuse} \oplus \mathsf{bind};(\mathsf{refuse} \oplus \mathsf{transform};(\mathsf{refuse} \oplus \mathsf{verify}^{+};(\mathsf{refuse} \oplus \mathsf{collapse}) \oplus \mathsf{verify}^{-};\mathsf{refuse}))\big),$$

where verify^{+} and verify^{-} are the two outcomes of verification, and after a negative outcome the only continuation is a refusal.

Theorem 15.10 (Protocol fidelity)**.** *Along every execution from a well-typed configuration, the sequence of actions whose subject is c is a trace of $\mathcal{S}$.*

Proof. By Theorem 15.5, each step updates the linear context by the context step, which follows exactly one edge of the table for the subject of the action. The first action on c is its pop or compensating pop. □

This is fidelity for the protocol of a single commitment. Protocols that coordinate several principals over channels, with a protocol and its dual held by different parties, need a calculus with communication; the correspondence with the session-type literature, including Wadler's propositions-as-sessions reading, is taken up when that extension is made.

15.5 Negative derivations

The type system's value is measured by what it rejects.

Proposition 15.11 (Rejected programs)**.** *There is no typing derivation for any of the following, in any context in which the named commitments' rights are as described.*

(a) A collapse of a commitment whose right is open, bound, or transformed.

(b) A bind of a commitment that has been refused: only a token and a fact remain, and T-Bind *requires a right.*

(c) Two uses of one right: either in parallel, since $\Delta_1 \uplus \Delta_2$ *cannot contain two rights for* c*, or in sequence, since after the first use the right has moved to a different state.*

(d) Verifying c_1 *with a verifier indexed by* $c_2 \neq c_1$*:* T-Ver *requires* $\mathsf{Verifier}\langle c_1, B\rangle$*.*

(e) Any attributed prefix without the corresponding authority fact.

(f) A process that ends holding a live right: T-Nil *admits only tokens.*

Proof. Each by Lemma 15.1: the only rule for the prefix in question has a premise the described context cannot satisfy. □

Case (c) is the static half of terminal exclusivity. Operationally, two components racing to refuse and collapse the same commitment produce exactly one terminal record (Chapter 9); statically, a well-typed program cannot contain the race, because only one component can hold the right.

Chapter 16

Authority as a Modality

The core calculus treats authority as a fixed policy: a set of facts $\mathsf{may}(a, \kappa, x)$ that no rule changes. That is enough for the safety theorems, and it makes non-amplification trivial. Real systems transfer authority, and a transfer is exactly the kind of event that must be recorded, authenticated, and replayable. This chapter adds delegation as a transfer of an exclusive capability, proves that it conserves authority rather than copying it, and then applies the resulting distinctions to a contemporary case.

16.1 Exclusive authority as a linear capability

Two kinds of authority are distinguished. *Shared* authority is a policy fact $\mathsf{may}(a, \kappa, x)$ in Γ, as before; any number of principals may hold it and nothing transfers it. *Exclusive* authority is a linear capability

$$\mathsf{Cap}\langle a, c, \kappa\rangle$$

held in Δ: the right of principal a, and of no one else, to perform acts of kind κ on commitment c. At runtime the configuration carries an ownership map Own from pairs (c, κ) to the single principal who currently holds the exclusive capability. An attributed rule whose authority premise was $\mathsf{may}(a, \kappa, c)$ now accepts either the shared fact or current ownership, $\mathsf{Own}(c, \kappa) = a$. Exercising a capability does not consume it; the typing rules pass it on to the continuation unchanged, so linearity forbids duplication and discarding without forbidding use.

Exclusive capabilities enter the world only through a grant record $\mathsf{Grant}(g, a, c, \kappa, \sigma)$ issued by a root authority g, and move only through delegation.

16.2 Delegation

The prefix $\mathsf{delegate}_{a\to b}\langle c, \kappa\rangle.P$ transfers a's exclusive capability over (c, κ) to b. Its typing rule consumes the delegator's capability and gives the continuation the delegate's:

$$\frac{\Sigma; \Gamma; \Delta, \mathsf{Cap}\langle b, c, \kappa\rangle \vdash P}{\Sigma; \Gamma; \Delta, \mathsf{Cap}\langle a, c, \kappa\rangle \vdash \mathsf{delegate}_{a\to b}\langle c, \kappa\rangle.P}.$$

Its operational rule checks current ownership and an authentication by the delegator, moves ownership, and appends an authenticated record:

$$\frac{\mathsf{Own}(c,\kappa) = a \qquad \mathsf{checkSig}(\sigma, a, \langle b, c, \kappa\rangle)}{\langle \mathsf{delegate}_{a\to b}\langle c,\kappa\rangle.P, \ldots, \mathsf{Own}, H\rangle \xrightarrow{\mathsf{delegate}(a,b,c,\kappa,\sigma)} \langle P, \ldots, \mathsf{Own}[(c,\kappa)\mapsto b], H \cdot \mathsf{Delegate}(a,b,c,\kappa,\sigma)\rangle}.$$

Ownership belongs to the replay state, not beside it. The capability key (c, κ) is a key of the replay map A in its own right, alongside the commitments, and its value is the current owner:

$$\begin{aligned} \delta(\bot, \mathsf{Grant}(g, a, c, \kappa, \sigma)) &= \mathsf{owner}(a), \\ \delta(\mathsf{owner}(a), \mathsf{Delegate}(a, b, c, \kappa, \sigma)) &= \mathsf{owner}(b) \qquad \text{if } \mathsf{checkSig}(\sigma, a, \langle b, c, \kappa\rangle), \end{aligned}$$

and once (c, κ) has been granted, replay accepts an act of kind κ on c only from the principal it holds as owner. Grant and delegation records write the capability key and belong to its projection; an exercise of a grantable kind reads it, before a grant as well as after (Definition 20.9). So $\mathrm{replay}(H) = \langle A, O\rangle$ keeps its form, with Own the restriction of A to capability keys, and the ledger path lemma holds for capability keys by the same argument as for commitments. Replay accepts $\mathsf{Delegate}(a, b, c, \kappa, \sigma)$ only if, at that point in the ledger, a owns (c, κ), the signature authenticates the record, and the commitment and kind match the signed payload. Because ownership moves, a delegation cannot be replayed a second time by its former owner: after it, a no longer owns what it would be delegating. Scope and expiry constraints, where a policy uses them, are further conjuncts of the same check.

Placing ownership in the replay map has a consequence worth stating now, because Chapter 24 depends on it. The principal who performed a completed act is part of the history only; the principal who currently holds an unconsumed exclusive capability is part of the state, and every future authority check consults it.

16.3 Results

Write $\mathsf{Owners}_H(c, \kappa)$ for the set of principals holding the exclusive capability for (c, κ) after replay of H.

Proposition 16.1 (Delegation authenticity)**.** *If replay of H is defined and H contains $\mathsf{Delegate}(a, b, c, \kappa, \sigma)$, then at that point in H the delegator a owned (c, κ) and σ authenticates a's transfer to b.*

Proof. These are the conditions under which replay accepts the record. □

Proposition 16.2 (Conservation)**.** *For every (c, κ) that has been granted, $|\mathsf{Owners}_H(c, \kappa)| = 1$ after every ledger prefix, and a delegation from a to b changes it by*

$$\mathsf{Owners}_{H'}(c, \kappa) = \big(\mathsf{Owners}_H(c, \kappa) \setminus \{a\}\big) \cup \{b\}.$$

In particular delegation never increases the number of holders of an exclusive capability.

Proof. A grant sets the owner of an ungranted pair. A delegation is accepted only from the current owner and replaces that owner by the delegate. No other record changes Own. □

Proposition 16.3 (Preservation under transfer)**.** *Extend well-typed configurations with the condition* $\mathsf{CapsAgree}(\mathsf{Own}, \Delta)$*:* $\mathsf{Cap}\langle a, c, \kappa\rangle \in \Delta$ *exactly when* $\mathsf{Own}(c, \kappa) = a$*. Then Theorem 15.5 holds for the calculus with delegation.*

Proof. The new case is delegation. Inversion of its typing rule gives the delegator's capability in Δ and types the continuation with the delegate's; the operational rule moves ownership from a to b, so $\mathsf{CapsAgree}$ is maintained, and the rights, commitment map, and history conditions are untouched except for the appended record, which replay accepts. Every other case is as before, with the authority premise now discharged either by a shared fact or, by $\mathsf{CapsAgree}$, by a capability in the context. Context splitting keeps each capability in exactly one component, by the same ownership argument as for rights (Lemma 14.1). □

Proposition 16.4 (Chain soundness)**.** *If an attributed record in H was authorized by an exclusive capability, there is a chain*

$$\mathsf{Grant}(g, a_0, c, \kappa, \sigma_0)\ ;\ \mathsf{Delegate}(a_0, a_1, c, \kappa, \sigma_1)\ ;\ \cdots\ ;\ \mathsf{Delegate}(a_{n-1}, a_n, c, \kappa, \sigma_n)$$

of authenticated records earlier in H, with a_n the principal of the record.

Proof. By induction on the ledger, ownership of (c, κ) at any point is the end of the chain of records that set it, each accepted only from the previous owner (Propositions 16.1 and 16.2). □

Together these give non-amplification in the presence of delegation: after any execution, the principals holding authority of kind κ over c are those holding the shared fact, which no rule adds, together with the single current owner of the exclusive capability, whose authority is traced to a grant by an authenticated chain.

Confinement. The results depend on capabilities having no other source. If the lambda layer could produce a value of capability type, for instance a function returning $\mathsf{Cap}\langle b, c, \kappa\rangle$ from nothing, conservation would fail at once: a transformation could mint authority. The calculus therefore gives capabilities no expression form. They appear only in linear contexts, introduced by grants and moved by delegation, and no value typing rule mentions them. That is a design constraint the propositions rely on, and any extension that lets computation handle capabilities as data must re-establish it.

What is left out. Revocation is not in the core. A revocation can race with the exercise of the capability it revokes, and deciding which happened first requires a declared linearization point, the same obligation that operational terminal exclusivity places on the sequencer (Hypothesis 9.4). It is a later extension.

16.4 Case study: evidence production is not disposition authority

A recent study of proposal selection at a large neutron-scattering facility gives a clean contemporary case for the distinctions this chapter formalizes [25]. The authors used a language model to compare every pair of proposals in a review cycle, aggregated the pairwise preferences into a ranking with a Bradley–Terry model, and compared the result with the historical

human rankings. Across cycles the rank correlations ranged from roughly 0.2 to 0.8, typically around 0.6, and proposals on which the two rankings diverged sharply were flagged for the review committee. They also used embedding similarity between proposals as a cheap triage signal for possible duplicates, and they are explicit that this signal is not a validated redundancy classifier: a high-similarity pair may be a resubmission, related work, or merely shared vocabulary, and telling these apart needs expert judgement. Their stated position is that the model should augment the committee rather than replace it, and that the committee keeps final authority over flagged divergences.

Two features of the evaluation bear on authority as well as on accuracy. Publication outcomes, used to compare the rankings' effectiveness, exist only for accepted proposals, and acceptance was historically decided by the human ranking; the authors note that this biases the comparison toward the human ranking. And where either ranking had ties, the other method's score was used as a secondary sorting key, so the two rankings are not wholly independent in tied cases. Neither point undoes the feasibility result. Both illustrate that evidence about an evidence producer's reliability can itself be shaped by earlier dispositions, which is exactly the kind of dependence a ledger makes auditable.

The study involves four roles, and the calculus can keep them apart:

- a *signal producer*, such as the embedding-similarity score, which flags candidates for attention and has no evidential standing;
- an *evidence producer*, such as the model's pairwise comparisons aggregated by Bradley–Terry, which produces a ranking with provenance that others may consult;
- a *verifier*, which checks a claim against an independent truth condition;
- a *disposition authority*, which accepts or refuses.

In the study the model is an evidence producer. Pairwise preference is not verification of scientific merit against an independent condition, and nothing in the study treats it as such. A committee may choose to verify the ranking *procedure*, for instance by checking position-swap consistency, but that is a verification of the procedure, performed under the committee's authority, not the model verifying the proposals.

Each role corresponds to a class of records it is able to write:

Role	Authority kind	Records it can write
signal producer	(annotation)	annotation-channel entries only; nothing causal
evidence producer	discharge	Discharged records; lifecycle-neutral
verifier	verify	Verify and VerificationFailed records; never terminal
disposition authority	collapse, refuse	Collapse and explicit Refuse records

Proposition 16.5 (Non-derivability of authority)**.** *In the calculus with a static policy, no transition and no typing rule adds an authority fact, and every attributed record requires the authority kind* $\mathsf{auth}(\alpha)$ *of Chapter 9. Hence a principal whose facts are exactly those of one role writes, in every execution, only records in that role's row of the table. In particular*

$$\mathsf{signal} \not\Rightarrow \mathsf{evidence}, \qquad \mathsf{evidence} \not\Rightarrow \mathsf{verify}, \qquad \mathsf{verify} \not\Rightarrow \mathsf{refuse}, \qquad \mathsf{verify} \not\Rightarrow \mathsf{collapse}.$$

Proof. The policy is a fixed set of facts consulted only by membership, and no rule's conclusion adds to it. Theorem 9.9 gives the required kind for each attributed record, and the table lists, for each kind, the records whose required kind it is. □

These are non-derivability results from the rules, not empirical claims that a model can never hold a later role. A policy may explicitly grant verification authority to a model. What the proposition rules out is authority arising merely because a component produced a score or a ranking. With delegation the same holds in stronger form: a principal can come to hold exclusive authority only at the end of an authenticated chain of grants and delegations (Proposition 16.4), each of which is an explicit, recorded act by the previous holder.

The fourth relation depends on a design decision taken for exactly this reason. In an earlier version of the calculus a failed verification refused its commitment automatically, which gave every verifier a refusal power through the negative branch even though the type system denied it explicit refusal authority. Verification now leads to $\mathsf{verificationFailed}$, a live state, and only a principal holding refusal authority can end the commitment from there. Verification classifies evidence; disposition determines continuation. The cost is one more live state, and one more way for a commitment to wait: a failed commitment whose disposition authority never acts stays live, and Chapter 23 lists that as a counterexecution rather than hiding it.

The encoding. Give the model only discharge authority. Each proposal awaits an obligation ω_{rank}, the model's ranking, which the model discharges with an authenticated certificate; the committee holds verification, collapse, and refusal:

$$\begin{aligned}&\mathsf{pop}\ q\ \mathsf{as}\ c.\ \mathsf{bind}_s\ c\ \mathsf{to}\ \mathit{proposal}.\ \mathsf{await}_m\ c\ \omega_{\mathrm{rank}}.\ \mathsf{transform}\ c\ \mathsf{with}\ \mathit{summary}.\\ &\quad \mathsf{verify}_h\ c\ \mathsf{using}\ \mathit{review}\ \mathsf{then}\ \mathsf{collapse}_{h,\mathrm{alloc}}\ c.\mathbf{0}\ \mathsf{else}\ \mathbf{0},\end{aligned}$$

with policy $\mathsf{may}(s, \mathsf{bind}, c)$ for the intake principal s, $\mathsf{may}(m, \mathsf{discharge}, \omega_{\mathrm{rank}})$, and $\mathsf{may}(h, \kappa, c)$ for $\kappa \in \{\mathsf{verify}, \mathsf{collapse}, \mathsf{refuse}\}$. A flagged divergence becomes a second obligation on which the committee's own verification waits.

Corollary 16.6 (Signals and evidence do not dispose)**.** *In every execution of this encoding, every record attributed to m is a discharge record, and no record attributed to m changes any commitment's lifecycle state.*

Proof. Proposition 16.5 for the evidence-producer row, with the discharge clause of δ returning the incoming state. □

Provenance of the ranking. A rank is a number, and a number looks more determinate than the process that produced it. The discharge certificate for ω_{rank} should therefore name, in the claim it hashes, everything the rank depends on: the comparison set, the model and prompt versions, the ordering controls used against positional bias, the aggregation parameters, and whether a human score was used as a tie-breaker. The last item matters for this study in particular, since its tie-breaking rule made the two rankings partly dependent. With those fields in the certificate, a later reader of the ledger can tell which ranking the committee saw and how it was made; without them, the ledger records that a ranking was delivered but not what it was evidence of.

In a typed program the separation is static as well: any prefix in which m binds, verifies, collapses, or refuses fails to type, since each rule requires the corresponding authority fact. The

committee's action supplies the attributable disposition. If the model's ranking were treated as the decision itself, synthetic evidence would have become synthetic authority, and the ledger would show exactly where: a terminal record attributed to a principal that should not have held the authority to write it. The general lesson is that when a system grants an automated component authority, the question is not what the component is meant to do but which records it is able to write.

Part V

The Disposition Calculus

Chapter 17

The Ledger Path Lemma

The theorems of this Part all concern what a history must contain once the commitment map says something about a commitment. They share one proof. Rather than repeat the same induction five times, this chapter proves a single invariant relating the ledger to the map, and the next three chapters read the individual results off it.

The projection $H|_c$, the subject of a record, and the lifecycle step function δ were defined in Section 1.4. The whole chapter turns on one feature of δ worth restating: it is a partial fold, not a filter. Every clause either advances the lifecycle or, for a discharge, checks authenticated evidence against a live state; none passes a record through unexamined. So if $\delta^*(H|_c)$ is defined, every record of $H|_c$ was consumed by exactly one step of the fold. A history cannot fold correctly while carrying a record the fold ignored.

17.1 The lemma

Call an initial configuration $C_0 = \langle P_0, S_0, A_0, O_0, H_0 \rangle$ *ledger consistent* when $\delta^*(H_0|_c) = A_0(c)$ for every $c \in \mathrm{dom}(A_0)$ and $H_0|_c$ is empty for every $c \notin \mathrm{dom}(A_0)$. The empty configuration, with A_0, O_0, and H_0 all empty, is ledger consistent.

Lemma 17.1 (Ledger path lemma)**.** *If C_0 is ledger consistent and $C_0 \to^* C = \langle P, S, A, O, H \rangle$, then for every commitment identity c:*

(i) $c \in \mathrm{dom}(A)$ if and only if $H|_c$ is nonempty;

(ii) if $c \in \mathrm{dom}(A)$ then $\delta^*(H|_c)$ is defined and equals $A(c)$.

Proof. By induction on the length of the execution. The base case is ledger consistency. For the step $C \xrightarrow{\alpha} C'$, history monotonicity gives $H' = H \cdot \rho$ with $\rho = \mathrm{record}(\alpha)$, and let $s = \mathrm{subj}(\rho)$.

For $c \neq s$, we have $H'|_c = H|_c$ because ρ is not in the projection, and $A'(c) = A(c)$ with $c \in \mathrm{dom}(A') \iff c \in \mathrm{dom}(A)$ by Theorem 9.5. Both clauses carry over.

For $c = s$, we have $H'|_s = H|_s \cdot \rho$ and must show $\delta(\delta^*(H|_s), \rho) = A'(s)$, where the inner fold is taken to be $\bot$ when $H|_s$ is empty. We examine the primitive rule at the root of the derivation.

- *Pop or compensating pop.* The premise $s \notin \mathrm{dom}(A)$ and clause (i) give $H|_s$ empty, so the fold is $\bot$, and δ returns $\mathsf{open}(q) = A'(s)$. Clause (i) now holds since $H'|_s = \langle \rho \rangle$.

- *Bind.* The premise $A(s) = \mathsf{open}(q)$ and clause (ii) give $\delta^*(H|_s) = \mathsf{open}(q)$, and δ applied to $\mathsf{Bind}(a, s, b)$ returns $\mathsf{bound}(q, b) = A'(s)$.
- *Transform.* From $A(s) = \mathsf{bound}(q, b)$ the fold yields that state, the record carries the same b because the rule reads it from $A(s)$, and δ returns $\mathsf{transformed}(b, v)$.
- *Verification, success.* From $A(s) = \mathsf{transformed}(b, v)$, the record carries that v, the premise $\mathrm{valid}(w, s, v)$ satisfies the side condition, and δ returns $\mathsf{verified}(v, w)$.
- *Verification, failure, and explicit refusal.* Both premises place s in a live state, satisfying the side condition, and δ returns $\mathsf{refused}(r)$ with the recorded r.
- *Discharge and timeout.* The premises give a live state and an authentic certificate, which are the side conditions of the corresponding clause, and δ returns the same state, as the rule leaves $A(s)$ unchanged.
- *Collapse.* From $A(s) = \mathsf{verified}(v, w)$, the record carries that w, the premise $o = \mathsf{observe}_k(v)$ satisfies the side condition, and δ returns $\mathsf{collapsed}(o)$.

In each case the rule's premises are precisely the conditions under which δ is defined on the new record, and its conclusion is precisely δ's value. □

The proof has a shape worth noticing. Nothing in it is specific to refusal, binding, or collapse; it only checks that each operational rule and the corresponding clause of δ say the same thing. That is the sense in which the rest of this Part is "nearly forced by the preceding architecture". The substance was spent in designing the rules so that each premise is a check on the ledger and each conclusion is a record.

17.2 Causal order

The ancestry theorems speak of one record causally preceding another, $e \prec_H e'$, and the spine asks that ledger order count only "where order represents a true dependency". The path lemma supplies the dependency for records of the same commitment: each record in $H|_c$ could be appended only because the previous one had produced the state its rule required. For records of different commitments, Chapter 20 shows that ledger order is generally not a dependency at all, since adjacent records of distinct subjects may be exchanged without changing anything replay can recover. We therefore take $\prec_H$ to be the dependence order of Definition 20.9, and note that every ancestry chain proved in this Part lies within a single projection $H|_c$, where ledger order and $\prec_H$ agree.

Chapter 18

Consequences of the Ledger Path Lemma

The ledger path lemma says that each commitment's projection is a legal path of δ from $\bot$ to its current state. Every local safety property of the untyped calculus is a reading of that fact at one particular state, and this chapter derives them in turn: refusal preservation, binding integrity, witness ancestry, and collapse soundness with terminal exclusivity. The first three are short and are given in proof-note form. Collapse soundness is the principal theorem, and it is followed by an analysis of its boundary: what it does not claim, and which premise each of its conclusions rests on.

Throughout this chapter C_0 is ledger consistent, $C_0 \to^* C = \langle P, S, A, O, H\rangle$, and Hypothesis 1.4 holds. Write $\mathsf{Open}(c, q)$ for either opening record, $\mathsf{Pop}(c, q)$ or $\mathsf{Compensates}(a, c, q, d)$.

18.1 Refusal preservation

Refusal must be treated as a successful terminal disposition, not as a missing execution.

Theorem 18.1 (Refusal preservation)**.** *If $A(c) = \mathsf{refused}(r)$ at some configuration C_i of an execution, then at every later configuration C_j,*

$$A_j(c) = \mathsf{refused}(r) \qquad \textit{and} \qquad \mathsf{Refuse}(a, c, r) \in H_j$$

for the principal a that refused it.

Proof. The first equation is lifecycle monotonicity, since $\mathsf{refused}(r)$ is $\preceq_D$-maximal and is related only to itself, together with the observation that no rule rewrites the reason of a terminal state. For the second, the path lemma at C_i gives $\delta^*(H_i|_c) = \mathsf{refused}(r)$; the only clause of δ producing that value is the refusal clause, so the last record of $H_i|_c$ is $\mathsf{Refuse}(a, c, r)$ for some a; and history monotonicity carries it into every H_j. □

The spine stated refusal preservation with an exception for a compensating successor. In this calculus the exception is not needed: compensation introduces a new identity and leaves the predecessor's state untouched (Theorem 9.5), so the refused commitment remains refused forever and it is the *successor* that carries the matter forward. A later acceptance cannot erase the refusal because there is no rule that could even address it.

18.2 Binding integrity

A binding should establish a durable relation without confusing association with identity.

Corollary 18.2 (Binding integrity)**.** *If* $A(c) = \mathsf{bound}(q, b)$*, then*

$$H|_c = \langle \mathsf{Open}(c, q),\ \mathsf{Bind}(a, c, b) \rangle$$

for some principal a*. In particular the binding record is unique, it is preceded by the opening record of the same commitment, and it binds the value* b *held in the state.*

Proof. By the path lemma, $\delta^*(H|_c) = \mathsf{bound}(q, b)$. Inspecting δ, the only record sequences folding from $\bot$ to a bound state are an opening record followed by one binding record. □

Commitment identities and expression values are disjoint sorts (Chapter 5), so c and b cannot become interchangeable because their concrete encodings coincide. The stronger clause of the spine, that an exclusive capability is bound to at most one live commitment, is not provable in the untyped calculus, where nothing marks a value as exclusive. It is a consequence of linearity and is proved in Part IV.

18.3 Witness ancestry

Corollary 18.3 (Witness ancestry)**.** *If* $A(c) = \mathsf{verified}(v, w)$*, then*

$$H|_c = \langle \mathsf{Open}(c, q),\ \mathsf{Bind}(a, c, b),\ \mathsf{Transform}(c, f, b, v),\ \mathsf{Verify}(a', c, v, w) \rangle$$

for some q, a, b, f, a'*, and* $\mathrm{valid}(w, c, v)$ *holds.*

Proof. The only record sequences folding from $\bot$ to $\mathsf{verified}(v, w)$ are an opening, a binding, a transformation, and a verification, with the matching conditions of δ forcing the b of the binding to equal the input of the transformation and the output v of the transformation to equal the value verified. The side condition of the verification clause is $\mathrm{valid}(w, c, v)$, and by stability it still holds. □

The four records form a chain in $\prec_H$ because they lie in one projection. The spine's requirement that a witness "witnesses" the transformation is realized by indexing: w is valid for the pair (c, v), and v is the output recorded in $\mathsf{Transform}(c, f, b, v)$. A witness produced for another commitment, or for another value of this one, fails the matching conditions of δ and has no rule by which it could enter this chain. What indexing does *not* rule out is a verifier g that issues valid witnesses too liberally; the calculus guarantees provenance of evidence, not the quality of the predicate, and that limit is stated here so that no later chapter is read as claiming otherwise.

18.4 Collapse soundness and terminal exclusivity

This is the principal local safety theorem of the untyped calculus.

Theorem 18.4 (Collapse soundness)**.** *If* $A(c) = \mathsf{collapsed}(o)$*, then*

$$H|_c = \langle \mathsf{Open}(c, q),\ \mathsf{Bind}(a, c, b),\ \mathsf{Transform}(c, f, b, v),\ \mathsf{Verify}(a', c, v, w),\ \mathsf{Collapse}(a'', c, k, w, o)\rangle,$$

with $\mathrm{valid}(w, c, v)$ *and* $o = \mathsf{observe}_k(v)$*, and* $(c, k, o) \in O$*.*

Proof. By the path lemma and inspection of δ, the only sequences folding to $\mathsf{collapsed}(o)$ are a sequence folding to some $\mathsf{verified}(v, w)$ followed by a collapse record carrying the same w and satisfying $o = \mathsf{observe}_k(v)$. Corollary 18.3 applied to that prefix supplies the first four records and validity. The observation (c, k, o) was added to O by the collapse rule that appended the last record, and no rule removes observations. □

The theorem rules out each of the failures the spine names. There is no collapse *ex nihilo*, because the fold requires an opening record. There is no collapse from an unrelated witness, because δ requires the collapse record's witness to be the one held in the verified state, which Corollary 18.3 ties to this commitment and value. There is no collapse after refusal, because δ is undefined on any record following a refusal. And there is no collapse whose ancestry has been omitted, because the projection is a subsequence of an append-only history and the fold must pass through every stage.

Corollary 18.5 (Terminal exclusivity, history form)**.** *For every commitment* c*, the projection* $H|_c$ *contains at most one terminal record, and if it contains one, that record is last. In particular*

$$\mathsf{Refuse}(a, c, r) \in H \quad \Longrightarrow \quad \mathsf{Collapse}(a', c, k, w, o) \notin H.$$

Proof. By the path lemma $\delta^*(H|_c)$ is defined. Since δ is undefined on every record whose incoming state is terminal, no record follows a terminal record in a sequence on which δ^* is defined. □

Reconsideration therefore never produces two terminal records for one identity. It produces a chain of distinct identities, which is the subject of the next chapter.

Remark 18.6 (Which collapse rule)*.* The spine stated collapse soundness with a single condition $\mathrm{denotes}(v, o)$. Here the condition is $o = \mathsf{observe}_k(v)$ for the declared rule k, recorded in the ledger. This is where the disposition calculus meets the kernel's rule-indexed collapse, and it bears directly on the third comparative claim: two commitments with equal verified values may collapse under different rules to different observations, and the ledger says which rule produced which. Whether this is more than a relabelling of a family of observational equivalences is the question of Chapter 24.

18.5 The boundary of collapse soundness

Collapse soundness is easy to over-read. Its conclusion is a statement about a ledger, and each of its clauses rests on a particular premise of δ or on a particular hypothesis. This section separates what the theorem says from what it might be taken to say, and, for each premise, gives the smallest history showing what fails without it.

Validity is not truth. The theorem concludes valid(w, c, v): the witness is acceptable, under the verifier whose version it records, as evidence for the value v at commitment c. It does not conclude that v is correct about anything outside the calculus. A verifier that accepts too liberally produces valid witnesses for false values, and every theorem of this book continues to hold for them. Collapse soundness is a theorem about the provenance of evidence, and the quality of the evidence predicate is a separate question, answered, if at all, outside the calculus (see the remark after Corollary 18.3).

Soundness of collapse is not adequacy of the observation rule. The theorem concludes $o = \mathsf{observe}_k(v)$: the observation is what rule k computes from the verified value. Whether rule k is a good reading of v for the consumer's purpose, whether it loses information the consumer needed or rounds in a way the consumer did not expect, is adequacy of the rule, and the calculus neither states nor checks it. What it guarantees is that the ledger names the rule, so that an observer who disputes the rule can see which one was applied and re-apply another to the same v.

History exclusivity, operational exclusivity, static exclusivity. Three results are easily confused. Corollary 18.5 holds for every replay-defined ledger with no further hypothesis: replay rejects a second terminal record. Operational exclusivity, that no execution ever *produces* a second terminal record, needs Hypothesis 9.4; without it, two replicas that each read $\mathsf{verified}(v, w)$ can append

$$\mathsf{Collapse}(a, c, k, w, o) \quad \text{and} \quad \mathsf{Refuse}(a', c, r)$$

concurrently, and the merged ledger, seven records including the four of the verified path, is simply not replay-defined. That is the minimal countermodel for dropping atomicity: the history form survives, as a rejection, and the operational form fails. Static exclusivity, that no program even *contains* two components able to race, is the typed result of Corollary 13.3.

What each premise of δ buys. Consider variants of δ, each dropping one condition, and in each the smallest history the variant accepts and the actual δ rejects. Write V for the four-record verified path $\mathsf{Pop}(c, q)\,;\mathsf{Bind}(a, c, b)\,;\mathsf{Transform}(c, f, b, v)\,;\mathsf{Verify}(a', c, v, w)$.

- *Dropping commitment indexing*, so that verification checks valid(w, v) rather than valid(w, c, v). The countermodel is V with w issued for a different commitment c' with the same value. It has four records, and it breaks the conclusion that the witness was produced for this commitment. A certificate issued for one applicant's file then certifies any other file that happens to contain the same data.

- *Dropping value indexing*, so that verification checks valid(w, c). The countermodel is V with w issued for (c, v'), $v' \neq v$, again four records. The witness then certifies the commitment rather than the result, and a later transformation could not be distinguished from the one verified; in this calculus there is no later transformation, but the conclusion valid(w, c, v) fails.

- *Dropping witness matching at collapse*, so that the collapse record may carry any witness. The countermodel is $V\,;\mathsf{Collapse}(a'', c, k, w', o)$ with $w' \neq w$ and w' valid for nothing, five

records. The collapse record then names evidence with no ancestry in the ledger, and an auditor who checks the record against its own witness is misled.

- *Dropping the observation check.* The countermodel is V followed by a collapse whose observation o' differs from $\mathsf{observe}_k(v)$, five records in all, and the conclusion $o = \mathsf{observe}_k(v)$ fails.

- *Allowing records after a terminal state.* The countermodel is V followed by a correct collapse and then a refusal of c, six records in all, with two terminal records for one commitment: history exclusivity fails, and refusal preservation fails in the other order.

None of these is exotic. Each is the form a plausible simplification of δ would take, and each is caught by exactly one premise. The premises are independent: in each countermodel every other premise holds.

Stability of the observation rule. Hypothesis 1.4 fixes $\mathsf{observe}_k$ for the duration of an execution relative to recorded versions. If rule k is redefined between the collapse and a later audit, so that $\mathsf{observe}'_k(v) = o' \neq o$, then replay under the new rule rejects a collapse that was correct when it happened; if the rule is redefined so that it agrees with a wrong o, replay accepts a collapse that was not. The minimal countermodel is the five-record successful history itself, replayed under a changed rule. The practical reading is that a rule index names a version, and the ledger that records k records enough to re-run the rule that was actually used.

Atomic update. The collapse rule reads $\mathsf{verified}(v, w)$ and writes $\mathsf{collapsed}(o)$ in one step. Its soundness does not depend on atomicity, since the history form is checked by replay whatever the interleaving; its exclusivity does, as above. This division, soundness from the fold and exclusivity from the map, is the one that the replicated setting of Chapter 22 makes precise.

Chapter 19

Compensation

Compensation must add history rather than revise it. It is the formal boundary between historical reversibility, which the calculus forbids, and material repair, which it permits.

Theorem 19.1 (Compensating continuation)**.** *If* $C \xrightarrow{\mathsf{compensate}(a,c',q,c)} C'$*, then*

(i) $c' \neq c$*;*

(ii) $H'|_c = H|_c$ *and* $A'(c) = A(c)$*, so the terminal record of* c *is present in* H' *and unchanged;*

(iii) $\mathsf{Compensates}(a, c', q, c) \in H'$ *and* $H \sqsubseteq H'$*;*

(iv) $\mathsf{may}(a, \mathsf{compensate}, c)$ *held in* C*.*

Proof. The rule's premises give $c' \notin \mathrm{dom}(A)$ and $\mathsf{terminal}(A(c))$, so $c \in \mathrm{dom}(A)$ and (i) follows. Part (ii) is Theorem 9.5 together with the fact that the new record's subject is c'; by Corollary 18.5 and the path lemma the last record of $H|_c$ is terminal. Part (iii) is history monotonicity and (iv) is Theorem 9.9. □

The spine required that "no projection may entail that the original disposition never occurred". That clause can be given a precise form.

Definition 19.2 (Supersession)**.** *Write* $c \rightsquigarrow c'$ *when* $\mathsf{Compensates}(a, c', q, c) \in H$*. The* current successor *of* c *is the last element of the longest chain* $c \rightsquigarrow c_1 \rightsquigarrow \cdots \rightsquigarrow c_n$ *in* H*, if such chains are unique, and the set of chain endpoints otherwise.*

Proposition 19.3 (Supersession is a view, not a revision)**.** *The relation* $\rightsquigarrow$ *is acyclic. For every commitment* c *and every extension* $H \sqsubseteq H'$ *produced by execution,* $\delta^*(H'|_c) = \delta^*(H|_c)$ *whenever* c *was terminal in* H*.*

Proof. If $c \rightsquigarrow c'$ then the opening record of c' appears later in H than the opening record of c, since c was already in the domain when c' was introduced. Opening positions strictly increase along any chain, so no chain returns to its start. For the second statement, a terminal c admits no further record in its projection by Corollary 18.5, so $H'|_c = H|_c$. □

Ancestry that runs through compensation crosses projections: the record

$$\mathsf{Compensates}(a, c', q, c)$$

lies in $H|_{c'}$ while referring to c. The path lemma covers each projection separately, so the crossing needs its own statement. Write $c \triangleleft_H c'$ when $\mathsf{Compensates}(a, c', q, c) \in H$.

Lemma 19.4 (Compensation ancestry)**.** *Let H be the history of an execution from a consistent configuration. If $c \lhd_H c'$, then $H|_c$ ends in a terminal record, and that record precedes the compensation record both in H and in the dependence order $\prec_H$. Consequently the ancestry of any commitment is a finite alternating sequence*

$$\pi_{c_0} \lhd \pi_{c_1} \lhd \cdots \lhd \pi_{c_n},$$

where each π_{c_i} is the lifecycle path $H|_{c_i}$, each π_{c_i} with $i < n$ ends in a terminal record, and every record is $\prec_H$-after every record to its left.

Proof. At the configuration where the compensation record was appended, the rule's premise made $A(c)$ terminal, so by the path lemma the last record of $H|_c$ at that point was terminal, and it was already in the history. The compensation record refers to c, so it is dependent on every record of $H|_c$ (Definition 20.9), and hence $\prec_H$-after them. Records within one projection are $\prec_H$-ordered because they share a subject. Finiteness and the absence of repeats follow from acyclicity of $\lhd_H$ (Proposition 19.3). □

A projection that reports only current successors is legitimate; it is a choice of what to display. What no execution can produce is a history from which the predecessor's disposition has become unrecoverable, because that disposition is a function of $H|_c$ and $H|_c$ never changes again. The distinction between the two is exactly the distinction between a Collapse rule and an edit.

Chains need not be unique. Two principals, each authorized, may compensate the same refused commitment, producing two successors. The calculus records both and does not choose; whether a policy should forbid this, or a later collapse should arbitrate between successors, is listed in Appendix E.

Part VI

Replay, Progress, and Concurrency

Chapter 20

Replay

20.1 Replay as a fold

Replay reconstructs the disposition and observation state from the ledger. Its state is a pair $X = \langle A, O\rangle$ and it applies one record at a time:

$$\mathsf{apply}(\langle A, O\rangle, \rho) = \begin{cases} \langle A[s \mapsto \delta(A(s), \rho)],\ O \cup \{(s, k, o)\}\rangle & \text{if } \rho = \mathsf{Collapse}(a, s, k, w, o), \\ \langle A[s \mapsto \delta(A(s), \rho)],\ O\rangle & \text{otherwise,} \end{cases}$$

where $s = \mathrm{subj}(\rho)$ and $A(s)$ is read as $\perp$ when $s \notin \mathrm{dom}(A)$. For a compensation record $\mathsf{Compensates}(a, s, q, d)$, apply is additionally defined only when $A(d)$ is terminal, matching the operational premise. apply is undefined whenever δ is, so replay of a ledger that no execution could have produced fails rather than silently repairing it. Then

$$\mathrm{replay}_{X_0}(H) = \text{the left fold of } \mathsf{apply} \text{ over } H \text{ from } X_0.$$

Theorem 20.1 (Replay correspondence)**.** *Let C_0 be ledger consistent with* $\mathrm{replay}_{\langle \varnothing, \varnothing\rangle}(H_0) = \langle A_0, O_0\rangle$*. If $C_0 \to^* \langle P, S, A, O, H\rangle$, then*

$$\mathrm{replay}_{\langle \varnothing, \varnothing\rangle}(H) = \langle A, O\rangle.$$

Proof. By induction on the execution. For a step appending ρ with subject s, apply updates the map at s by δ, which by the path lemma agrees with the operational update, and leaves every other key unchanged, which agrees with Theorem 9.5. For a compensation record the additional condition on $A(d)$ is the operational premise. The observation component changes only on a collapse record, by the same triple the collapse rule adds to O. □

The store S is absent from the statement, and not by oversight. In the rules of Chapter 9 no transition changes S; the component is carried for the extensions in which transformations read and write mutable resources, receive external input, or behave nondeterministically. In those extensions S is reconstructible exactly when every state-affecting input is itself recorded, and not otherwise. The present theorem claims only the part that holds unconditionally.

Replay also serves as a checker. Because apply is partial, replay(H) is defined only on histories whose every projection is a legal lifecycle path. By the path lemma every history produced by execution passes this check, so a ledger received from elsewhere can be audited by replaying it: success certifies that each commitment's records form an admissible path, with valid witnesses, matching data, and at most one terminal record, provided Hypothesis 1.4 holds at the auditor.

20.2 Consistency as an invariant

The replay theorem is stated for executions from a ledger-consistent start. It is cleaner to state its content as an invariant on single configurations, since that is what a running or restored system needs.

Definition 20.2 (Consistency)**.** *A configuration* $C = \langle P, S, A, O, H\rangle$ *is* consistent*,* $\mathsf{Consistent}(C)$*, when* $\mathrm{replay}(H) = \langle A, O\rangle$.

Proposition 20.3 (Consistency is established and preserved)**.** *The empty configuration is consistent. If* $\mathsf{Consistent}(C)$ *and* $C \xrightarrow{\alpha} C'$*, then* $\mathsf{Consistent}(C')$.

Proof. The first statement is immediate. For the second, let ρ be the appended record with subject s. By consistency $A(s)$ is the replayed state of s, the rule's premises are exactly the conditions under which $\delta(A(s), \rho)$ is defined, and its conclusion writes that value. The map is unchanged elsewhere (Theorem 9.5), and the observation set changes only on collapse, by the triple replay adds. □

Theorem 20.1 is the iterate of this proposition. A system that starts empty is consistent forever. A system resumed from stored state is admitted only through a checked initialization, which combines the consistency test with the completeness anchor of the next section:

$$\frac{\mathrm{replay}(H) = \langle A, O\rangle \qquad \mathsf{ValidAnchor}(H, \chi)}{\mathsf{init}(A, O, H, \chi)\ \mathsf{ok}}.$$

One invariant then covers empty systems, restored systems, and embedded long-running systems alike, and the ledger path lemma applies to all of them.

20.3 Completeness and anchors

Replay as a checker certifies that each commitment's records form an admissible path. It cannot certify that the ledger is complete.

Theorem 20.4 (Prefix indistinguishability)**.** *If* H *is the history of an execution from a ledger-consistent configuration, then every prefix* $H' \sqsubseteq H$ *is itself the history of an execution from the same configuration, and* $\mathrm{replay}(H')$ *is defined. Consequently no predicate on histories alone separates complete ledgers from truncated ones.*

Proof. Stop the execution after its first $|H'|$ transitions. By history monotonicity its history at that point is H', and by Theorem 20.1 its replay is defined. A predicate that accepted H as complete and rejected H' as truncated would reject the complete history of this shorter execution. □

Truncation is only the simplest case. The laboratory found interior deletions that replay cannot detect either (Chapter 2), and the general statement is this.

Theorem 20.5 (Subhistory indistinguishability)**.** *There are histories* H *produced by execution and proper subsequences* H' *of* H*, not prefixes of it, such that* $\mathrm{replay}(H)$ *and* $\mathrm{replay}(H')$ *are both defined and* H' *is itself the history of an execution. Hence replay validity certifies the local admissibility of the records presented, not their completeness relative to what actually occurred.*

Proof. Take $H = \mathsf{Pop}(c,q)\,;\mathsf{Bind}(a,c,b)\,;\mathsf{Transform}(c,f,b,v)\,;\mathsf{Refuse}(a,c,r)$ and delete the transformation. The remainder $\mathsf{Pop}(c,q)\,;\mathsf{Bind}(a,c,b)\,;\mathsf{Refuse}(a,c,r)$ is the history of the execution that refuses from the bound state, which the explicit refusal rule permits. Both replay. □

Detection therefore needs information that is not in the history. The usual object is a hash chain with a signed checkpoint. Fix a hash function h and let

$$h_0 = \mathsf{init}, \qquad h_i = h(h_{i-1} \parallel e_i),$$

where e_i is the i-th ledger entry with its event identity. A *checkpoint* is a triple $\mathsf{Anchor}(n, h_n, \sigma)$ in which σ is the sequencer's signature on (n, h_n).

Proposition 20.6 (Relative completeness)**.** *Suppose an auditor has retained an authentic checkpoint* $\mathsf{Anchor}(n, h_n, \sigma)$ *independently of the ledger, the signature scheme is unforgeable, and h is collision resistant. If a presented ledger H has hash chain value h_n at position n, then its first n entries match the anchored sequence in length, order, and content, except with negligible probability. A presented ledger shorter than n, or with a different chain value at n, is detected, whether the discrepancy is a truncation, an interior deletion, an insertion, a reordering, or an altered record.*

Proof. The first two cases are direct checks. For the third, two different sequences of n entries hashing to the same h_n yield, at the first position where the chains diverge, a collision in h. The argument is the standard one for hash chains and is used here without further elaboration; its strength is exactly that of the cryptographic assumptions. □

The completeness obtained is relative: relative to the anchor, and to trust in whoever issued it or in the transparency mechanism that holds it. The hypothesis that carries the weight is *independent retention.* A sequencer that can replace both the ledger and the checkpoint can publish a shorter history with a fresh, internally consistent checkpoint, and nothing internal to the pair reveals it. Deployments close that gap by placing checkpoints where the sequencer cannot rewrite them: in an external append-only transparency log of the kind used for web certificates [26], or distributed among independent witnesses. The lesson fits the calculus closely. A history can vouch for the admissibility of what it contains; a claim that nothing is missing needs a witness outside the history whose completeness is in question.

20.4 Duplicate delivery

When records travel between replicas they may arrive more than once. Pair each record with an *event identity* i assigned by the sequencer when the record is appended, so that a ledger is a sequence of pairs (i, ρ). Extend the replay state with the set I of identities already applied, and let

$$\mathsf{apply}(\langle A, O, I\rangle, (i,\rho)) = \begin{cases} \langle A, O, I\rangle & \text{if } i \in I, \\ \langle \mathsf{apply}(\langle A, O\rangle, \rho),\ I \cup \{i\}\rangle & \text{otherwise.} \end{cases}$$

Hypothesis 20.7 (Identity coherence)**.** *Any two ledger entries with the same event identity carry the same record.*

Theorem 20.8 (Idempotent disposition)**.** *For every state X and entry e,* $\mathsf{apply}(\mathsf{apply}(X, e), e) = \mathsf{apply}(X, e)$. *Under Hypothesis 20.7, if* $\mathsf{dedup}(H)$ *keeps the first occurrence of each identity, then*

$$\mathrm{replay}(H) = \mathrm{replay}(\mathsf{dedup}(H))$$

on the $\langle A, O\rangle$ components, both sides being defined or undefined together.

Proof. The first statement is immediate, since after the first application $i \in I$. For the second, induct on H. An entry whose identity is new is applied on both sides. An entry whose identity was seen earlier is a no-op on the left and absent on the right; by coherence it is the same record as the earlier occurrence, so dropping it cannot change which occurrence was the one applied. □

The mechanism is deliberately simple; any system with an applied-set achieves idempotence. The content of the theorem is in its hypothesis. Exactly-once *disposition* is obtained from at-least-once *delivery* precisely when the sequencer never reuses an identity for a different record, and that is an obligation on the arbiter, not a property the replay procedure can establish for itself.

20.5 Which reorderings replay ignores

Duplicate tolerance concerns the same record arriving twice. A different question is whether distinct records may arrive in a different order.

Definition 20.9 (Independence and dependence order)**.** *Each record* writes *one key,* $\mathrm{wr}(\rho)$*, and* reads *a set of further keys,* $\mathrm{rd}(\rho)$*. Keys are commitments and, once exclusive authority is in use, capability keys (c, κ) (Chapter 16). A record writes its subject, or, for a grant or delegation, its capability key. A compensation reads its predecessor; an act of kind κ on c reads (c, κ) whenever κ is a grantable kind, whether or not a grant has yet occurred; no other record reads anything. The read is unconditional for a reason: an ungated exercise made before a grant must not commute past the grant, after which the same exercise by a non-owner would be rejected. Records ρ and σ are* independent*, written $\rho \, I \, \sigma$, when* $\mathrm{wr}(\rho) \neq \mathrm{wr}(\sigma)$, $\mathrm{wr}(\rho) \notin \mathrm{rd}(\sigma)$*, and* $\mathrm{wr}(\sigma) \notin \mathrm{rd}(\rho)$*: neither writes a key the other reads or writes. Without capability keys this says that the subjects differ and neither is a compensation of the other's subject. Two histories are* trace equivalent*, $H \sim_I H'$, when one can be obtained from the other by finitely many exchanges of adjacent independent records. The* dependence order $\prec_H$ *is the transitive closure of the relation "ρ precedes σ in H and ρ, σ are not independent".*

Proposition 20.10 (Replay respects commutation)**.** *If $H \sim_I H'$ then* $\mathrm{replay}(H)$ *is defined if and only if* $\mathrm{replay}(H')$ *is, and when defined they are equal.*

Proof. It suffices to treat one exchange, $H = H_1 \cdot \rho \cdot \sigma \cdot H_2$ and $H' = H_1 \cdot \sigma \cdot \rho \cdot H_2$ with $\rho \, I \, \sigma$. From the state reached after H_1, applying ρ writes the map at $\mathrm{wr}(\rho)$ and consults it only there and at $\mathrm{rd}(\rho)$. By independence, σ writes none of these keys, and symmetrically. So each application is defined, and produces the same update, whether it comes first or second. The observation components are unions of sets and commute. The remainder H_2 is then applied to equal states. □

Corollary 20.11. *The disposition and observation state recoverable from a ledger is a function of its Mazurkiewicz trace $[H]_{\sim_I}$. Every ancestry chain of Chapter 18 is a $\prec_H$-chain.*

Proof. The first sentence is Proposition 20.10. For the second, consecutive records of one projection share a subject and are therefore dependent. □

This result has two consequences for the rest of the book. It makes precise the spine's phrase "ledger order where order represents a true dependency": the true dependencies are those of $\prec_H$, and the total order imposed by the sequencer carries more information than replay can use. And it bears on the first comparative claim, since a trace with an independence relation determined by subjects is very close to the configuration structure of an event structure. Chapter 21 takes that comparison up directly, and Chapter 22 asks what the sequencer's extra information is for, given that replay does not need it.

Chapter 21

Petri Nets, Traces, and Event Structures

The first comparative claim of the Preface is that the calculus's nearest formal relative is the theory of event structures. This chapter settles it. It does so by building two event structures from the calculus, one from a single history and one from all the continuations a configuration admits, and proving that the second is a prime event structure whose configurations are exactly the executions up to trace equivalence. The residue that ordinary event structures do not carry is then isolated by a separation result.

The two constructions must be kept apart, because a single ledger contains only the branch that happened. Conflict describes alternatives that did not happen, and no ledger can record those. So the occurrence structure $\mathcal{O}(H)$ of one history has no conflict at all, while the branching structure $\mathcal{W}(C)$ of a configuration is where conflict lives.

21.1 The occurrence structure of a history

Definition 21.1 (Occurrence structure)**.** *For a history H with* replay(H) *defined, let*

$$\mathcal{O}(H) = \langle E_H, \leq_H, \#, \lambda_H, \eta_H \rangle,$$

where E_H is the set of positions of H (record occurrences); $\leq_H$ is the reflexive closure of the dependence order $\prec_H$ of Definition 20.9; the conflict relation is empty; $\lambda_H(i)$ is the action kind and subject of the record at position i; and $\eta_H(i)$ is the rest of the record: principal, request, value, witness, certificate, reason, observation rule, and obligation, as the record kind has them.

Events are occurrences rather than record values as a matter of principle, but within one replayable history the distinction is harmless: no record occurs twice (Lemma 3.4). Across branches it is not harmless, since the same record value can occur after different pasts, and the branching construction below identifies events by their pasts for that reason.

Theorem 21.2 (Trace invariance of occurrence structures)**.** *If $H \sim_I H'$ then $\mathcal{O}(H) \cong \mathcal{O}(H')$ by an isomorphism that preserves $\leq$, λ, and η, though not ledger positions.*

Proof. It suffices to treat one exchange of adjacent independent records at positions $i, i+1$. Map each position to the position of the same record in H', which swaps i and $i+1$ and fixes

the rest. Labels and annotations are preserved because the records are. Independence is a relation on records, so the pair $(i, i+1)$ is unrelated in both orders, and every other pair of records keeps its relative order, so the dependence relation, and with it its transitive closure, corresponds. □

This is the event-structural form of Proposition 20.10. Replay depends only on the trace, and the occurrence structure is the trace presented as a partial order.

21.2 The branching structure of a configuration

Fix a consistent configuration C with history H_0. Call a finite sequence of records K an *admissible extension* if $\mathrm{replay}(H_0 \cdot K)$ is defined. Replay here is the extended replay of Chapter 16: its map is keyed by commitments and, once exclusive authority is in use, by capability keys (c, κ), whose values record the current owner, and dependence is the read/write dependence of Definition 20.9. Without that extension a grant followed by a delegation would be invisible to replay, and two exercises of one capability by different principals could both appear admissible. This is the ledger-level semantics of Chapter 3, the behaviour of the universal process; a particular program realizes a subset of it.

Definition 21.3 (Branching disposition structure)**.** *An admissible extension K is* prime *if its last record is the unique maximal element of its dependence order. The events of $\mathcal{W}(C)$ are prime extensions up to trace equivalence. For an admissible extension K and a position i of K, write $\partial_K(i)$ for the event given by the $\prec_K$-downset of i, listed in any order compatible with $\prec_K$. Then*

- *$e \le e'$ when $e = \partial_K(i)$ and $e' = \partial_K(j)$ for some admissible K with $i \preceq_K j$;*
- *$e \# e'$ when no admissible extension contains both, that is, there is no K with $e = \partial_K(i)$ and $e' = \partial_K(j)$;*
- *$\lambda(e)$ and $\eta(e)$ are those of the last record of e.*

For an admissible K, write $\mathrm{ev}(K) = \{\partial_K(i) \mid i \text{ a position of } K\}$.

A record value may be the last record of several events, with different pasts: a binding record after an ordinary opening and the same binding record after a compensating opening are different events, and in conflict. This is the reason events are prime extensions rather than records.

Lemma 21.4 (Same key, same chain)**.** *Let X be a finite set of events that is downward closed and pairwise non-conflicting. Then any two events of X whose last records are dependent are $\le$-comparable. In particular, for each key x, a commitment or a capability key, the events of X whose last record writes x form a chain, and every event whose last record reads x is comparable with each of them.*

Proof. Take $e, e' \in X$ with dependent last records. They do not conflict, so some admissible K has $e = \partial_K(i)$ and $e' = \partial_K(j)$. The records at i and j are dependent, so they are $\prec_K$-ordered, say $i \prec_K j$, whence $e \le e'$. Two records that write the same key are dependent, and so are a record that writes a key and a record that reads it. □

Theorem 21.5 (Binary consistency)**.** *$\mathcal{W}(C)$ is a labelled prime event structure: $\leq$ is a partial order with finite downsets, and $\#$ is irreflexive, symmetric, and hereditary. Moreover, a finite set of events X is a configuration, downward closed and pairwise non-conflicting, if and only if $X = \mathrm{ev}(K)$ for some admissible extension K; and $\mathrm{ev}(K) = \mathrm{ev}(K')$ if and only if $K \sim_I K'$.*

Proof. Order and conflict. Every event is a finite prime extension, so its downset is finite. Antisymmetry holds because $e \leq e'$ makes e a prefix of e' up to trace equivalence, and prefixes in both directions are equal. Conflict is irreflexive and symmetric by definition. It is hereditary because if $e' \leq e''$ and some K contained e and e'', then K would contain the prefix e' of e'' too.

Configurations from executions. $\mathrm{ev}(K)$ is downward closed, since the downset of a downset is contained in it, and pairwise non-conflicting, since K contains all of it.

Executions from configurations. Let X be a configuration and list its events in any order compatible with $\leq$, taking the last record of each. Call the result K. We show $\mathrm{replay}(H_0 \cdot K)$ is defined by checking each clause of δ where it is applied. Fix a key x, a commitment or a capability key. By Lemma 21.4 the events of X that write x form a chain, so the x-projection of K lists them in chain order. The greatest of them, $e_{\max}$, is a prime extension whose own x-projection contains every earlier write of x it depends on, which is all of them; since each event of the chain is a prefix of $e_{\max}$, those records are exactly the x-writing events of X. So the x-projection of K is the x-projection of $e_{\max}$, which is a legal path because $e_{\max}$ is admissible.

It remains to check the clauses of δ that consult a key other than the one written. There are two: a compensation reads its predecessor d and needs it terminal, and an act of grantable kind κ on c reads (c, κ) and needs the key either ungranted or owned by its own principal. The argument is the same for both. Let e be the reading event and x the key read. The reading record depends on every write of x, so every write of x that preceded it in the admissible extension defining e is in e's past; by downward closure those writes are in X and precede e in K. Any other write of x in X is, by Lemma 21.4, comparable with e and not below it, so it lies above e and follows it in K. Hence at e's position in K the value of x is exactly the value it had when e was admissible: a terminal state of d in the first case, and either $\bot$ or $\mathsf{owner}(p)$ for e's principal p in the second. For compensation there is no later write, since terminal states are final; for capabilities there may be one, a delegation away from p, and the argument places it after the exercise. The settled-obligation sets are part of each subject's own state and are covered by the path argument. Hence K is admissible. Its events are X: the past of each position of K is the corresponding event's past, because by Lemma 21.4 dependent events of X are comparable, so the dependence order of K restricted to each event's downset is that event's own order.

Uniqueness up to trace. Two linearizations of X compatible with $\leq$ order every comparable pair alike and, by Lemma 21.4, every dependent pair is comparable, so they differ only in the order of independent records, which is trace equivalence. Conversely, trace equivalent extensions have the same downsets, hence the same events. □

The theorem's content is in the word *binary*. A set of events that is consistent pair by pair is consistent as a whole. That is not automatic: a constraint such as "at most two commitments may be refused" is consistent pair by pair and inconsistent in threes, and a calculus with such a constraint would need a general event structure with a consistency predicate on sets, not a prime one. The proof shows why the calculus has no such constraint: every clause of δ reads the path of the key it writes, together with at most one other key's current value, a predecessor's

terminality or a capability's owner, and each of these is fixed by the chain of writes to that key, which pairwise comparability already orders.

Corollary 21.6 (Claim 1, positive part)**.** *At the ledger level, the semantics of a configuration is a labelled prime event structure: its configurations are its executions up to trace equivalence, and the Causal Shuffle theorem (Theorem 2.9) is the statement that replay is a function on configurations.*

21.3 The residue

The labels of $\mathcal{W}(C)$ carry more than an ordinary event structure needs. Write

$$U\langle E, \leq, \#, \lambda, \eta\rangle = \langle E, \leq, \#, \lambda\rangle$$

for the reduct that forgets the annotations, keeping only action kinds and subjects.

Proposition 21.7 (Forgetting the annotations)**.** *There are histories H_1, H_2 with $U(\mathcal{O}(H_1)) \cong U(\mathcal{O}(H_2))$ and $\mathcal{O}(H_1) \ncong \mathcal{O}(H_2)$, which differ only in the observation rule of a collapse, and whose collapses produce the same observed value.*

Proof. Let $\mathsf{observe}_{k_1}(v) = \mathsf{observe}_{k_2}(v) = o$ for rules $k_1 \neq k_2$, and let H_i be the successful history of Chapter 9 ending in $\mathsf{Collapse}(a, c, k_i, w, o)$. Both reducts are a chain of five events with the same kinds and subject. Any isomorphism of the full structures must match the collapse events, whose annotations differ in the rule. □

The same construction separates histories that differ only in a refusal reason, a verification certificate, an acting principal, or the chain of delegations behind an act. The reduct therefore preserves causality, conflict, concurrency, and the kind of each act, and forgets exactly the material on which the book's guarantees rest: who acted and with what authority (Theorem 9.9), which evidence a verification rests on and for which commitment (Corollary 18.3), why a refusal was made, and under which rule an observation was produced (Chapter 9).

It would be a mistake to state the residue as "event structures cannot carry witnesses". They can carry any labels one likes. The accurate statement has two parts. First, the ordinary reduct forgets these distinctions, and the enriched structure keeps them. Second, the calculus supplies something an event structure does not: the rule that generates $\mathcal{W}(C)$. Which events exist depends on the annotations, through valid, $\mathsf{checkCert}$, $\mathsf{observe}_k$, and the authority policy, which are predicates over unbounded data. An event structure is a description of a behaviour; the calculus is a way of producing one from evidence and authority.

21.4 Petri nets

The same picture has a net reading. A single commitment is a state-machine net with one token: places are lifecycle state kinds and transitions are record kinds, with the data carried by states and records as colours, so that the net is coloured. Its unfolding in the sense of Nielsen, Plotkin, and Winskel is an occurrence net whose event structure is the part of $\mathcal{W}(C)$ with that subject.

Compensation adds one kind of arc between the nets of different commitments: the successor's opening transition tests the predecessor's terminal place without consuming its token.

Nets with such read arcs, contextual nets, unfold in general not to prime but to asymmetric event structures, because an event that reads a place must precede any event that later consumes the token there. That complication does not arise here. Terminal places are never consumed, since nothing follows a terminal state, so no reading event ever competes with a consuming one, and the prime structure of Theorem 21.5 suffices. The calculus sits, in net terms, at the simple end of the contextual-net spectrum, and it does so because terminality is final.

21.5 Bounded checks

The laboratory builds $\mathcal{W}$ from the empty configuration, with events computed as prime extensions and identified by a canonical linearization of their pasts, and checks four bounded propositions over the admissible ledgers of at most six records: every ledger's event set is a configuration (P-ESRealized); every configuration of at most six events replays (P-ESComplete); conflict is hereditary (P-ESHeredity); and all linearizations of each ledger yield the same events with the same order (P-ESTraceIso). The run uses laboratory version 5, whose replay includes capability ownership, grants, and delegations, with read/write dependence. The class has 11,032 ledgers, 605 events, 162,702 conflicting pairs, 1,819 configurations, and 334,871 linearizations, and all four hold.

The checker needed one correction worth recording. Its first version approximated compatibility of two events by asking only whether the union of their pasts replays. That is weaker than the definition: two events can have jointly replayable pasts while containing dependent records that neither past orders, so that no single execution contains both with the pasts they claim. With the approximation, P-ESHeredity failed on 418 pairs at five records. The corrected checker also requires every dependent pair in the union of the pasts to be causally ordered, which is what Definition 21.3 says, and heredity then holds. The failure was the checker's, not the theorem's, and it is recorded because it is the kind of error a bounded check is liable to: a plausible approximation of a definition that the definition does not license.

Two further observations came out of the run. As a negative control, adding the ternary constraint "at most two refusals" to replay, with three commitment identities available and the authority and settlement records disabled, makes P-ESComplete fail on eight of 774 configurations, as Theorem 21.5's discussion predicts; the check detects the failure of binary consistency it exists to detect. And only 1,622 of the 1,819 configurations are realized by a ledger the enumerator produced. The rest, such as the single event $\mathsf{Pop}(c_2)$, are perfectly admissible but use a commitment name the enumerator, which always picks the least fresh name, never chooses first. The event structure is closed under renaming where the enumerator is not, which is a small, concrete instance of why events should be identified up to trace and name rather than by the ledger that happened to produce them.

Chapter 22

Logical Clocks, CRDTs, and the Single Sequencer

The kernel assumes a single sequencer, an arbiter that assigns every event a place in one total order. Replay does not need that order (Proposition 20.10), and the event-structure semantics does not contain it (Theorem 21.5). This chapter asks what the total order is for, and answers with a theorem: typed programs whose commitments are statically owned converge under replication without any replica waiting for agreement, and every obligation that does need agreement can be named.

22.1 Happened-before

Lamport's happened-before relation on a distributed execution orders two events when they occur in sequence at one site or are linked by a message. The dependence order $\prec_H$ of a ledger is the part of happened-before that replay consults: two records are ordered by it exactly when one writes a key the other reads or writes (Definition 20.9), that is, when they concern the same commitment, one is a compensation of the other's commitment, or they touch the same capability key. Any linear extension of happened-before is therefore a linearization of the dependence order, and by the Causal Shuffle theorem every such linearization replays to the same state. A Lamport clock, which yields one such linear extension, is enough to produce a ledger an auditor can replay; nothing finer is needed for that purpose.

22.2 Convergence without consensus

Consider an execution distributed over replicas $R_1, \ldots, R_m$. The word *coordination* is used here in a local, checkable sense.

Definition 22.1 (Coordination-free production)**.** *A replicated execution is* coordination free *when no replica, in order to append a record, waits for a message from or agreement with any other replica. Communication is limited to dissemination: a replica sends the records it appends and applies the records it receives, and neither step blocks the production of new records at the sender.*

Causal delivery may delay the *application* of a received record, but not the production of any record, so it is compatible with the definition.

Definition 22.2 (Static ownership)**.** *A replicated execution has* static ownership *when each commitment c has an owner replica* $\mathrm{own}(c)$*, fixed when c is introduced, such that*

(i) every record with subject c, and every grant or delegation of a capability key (c, κ), is appended first at $\mathrm{own}(c)$*, in the order of a single execution there; that is,* $\mathrm{own}((c, \kappa)) = \mathrm{own}(c)$*;*

(ii) each replica draws new commitment names and event identities from its own namespace, for instance pairs of the replica's name and a local counter;

(iii) records are disseminated with causal delivery: a replica applies a received record only after every record it depends on in the dependence order.

Typed programs fit the definition naturally. A commitment's right is held by one occurrence in the program text (Lemma 14.1), so placing that occurrence at a replica makes the replica the commitment's owner, and a pop executed there creates a commitment it owns.

Theorem 22.3 (Coordination-free convergence under static ownership)**.** *Under static ownership, the execution is coordination free in the sense of Definition 22.1, and at every moment:*

(a) each replica's local ledger, the records it has applied in the order it applied them, has a defined replay;

(b) two replicas that have applied the same set of records have the same replay state; and

(c) that state is the replay of any linearization of the global dependence order restricted to that set.

Proof. Let X be the set of records a replica has applied. By causal delivery X is downward closed in the global dependence order, and the replica applied them in an order compatible with it: records writing one key, a commitment or one of its capability keys, were produced in order at a single owner and are delivered in that order, and a record that reads a key, a compensation or an exercise of a granted kind, is applied only after the writes of that key it depends on. The global execution's ledger is replay-defined, and X is a downset of its dependence order, so some linearization of the global ledger begins with a linearization of X, whose replay is defined as a prefix of a defined fold. Every linearization of X is trace equivalent to that one, since they agree on all dependent pairs, and so replays identically (Proposition 20.10). This gives (a) and (c), and (b) follows from (c). Freshness and identity coherence hold without coordination because each replica draws from its own namespace. Coordination freedom holds because every check a record must pass at production time consults only keys owned by the producing replica, or, for compensation, the terminal state of a predecessor, which the producer has already applied and which no later record can change. □

In the vocabulary of replicated data types, the ledger under static ownership is a grow-only set of records whose merge is union, and convergence follows from the independence of records of distinct commitments rather than from any property of a merge function. It is also an instance of the principle, stated by Hellerstein and Alvaro, that programs whose outputs grow

monotonically with their inputs need no coordination. A ledger only grows, a commitment's state is a function of its own records, and the only read across replicas, compensation's test of a predecessor's terminality, is of a fact that never becomes false. The reads of capability keys are not monotone, since ownership moves, but they are local: the key and every exercise that reads it live at the same owner replica.

22.3 What needs coordination

The theorem locates the need for coordination exactly, by listing what static ownership rules out.

- *Two replicas acting on one commitment.* Operational terminal exclusivity relies on the atomic commitment map (Hypothesis 9.4). Under static ownership that hypothesis is local to the owner, where it holds trivially. Without it, two replicas can each read a live state and append incompatible terminal records, and some consensus object, a lease, or a single sequencer for that commitment is needed. The typed calculus never requires this, since only the owner of a right can act on its commitment; the untyped race of Chapter 13 does.

- *Moving ownership.* Failover that transfers a commitment to a new owner is a handoff that two replicas must agree on, and it is a consensus problem in general.

- *Delegation* does not need consensus. A delegation record is appended at the owner replica of the commitment, where the capability key lives, and every later use of the capability reads that key at the same replica, so the check is local.

- *Competing compensations* do not need consensus either. Two successors of one terminal commitment have distinct subjects and are independent of each other; both are admitted, and whether a policy should forbid this is a question about policy, not about consistency (Appendix E).

- *Completeness anchors* need a sequence to hash, but not a global one. Each owner can chain its own records, and a checkpoint can consist of the vector of per-replica chain heads (Proposition 20.6).

The two kinds of ownership in the delegation bullet should not be confused. *Principal delegation* moves the exclusive capability for (c, κ) from one principal to another; it is a record like any other, appended where c lives. *Replica ownership transfer* moves the place where records of c are produced; it changes $\mathrm{own}(c)$ and leaves static ownership. The first is coordination free because it does not change which replica checks it. The second is not, because two replicas must agree on which of them now checks. A delegate principal who sits at another replica submits its exercises to $\mathrm{own}(c)$; it does not thereby become the replica that owns c.

What the total order of the kernel's single sequencer provides, beyond these, is a presentation: one ledger that can be handed to an auditor. By the Causal Shuffle theorem any linearization serves equally well, so the sequencer's order is a convenience of presentation, not part of the semantics.

22.4 Outcome

The purpose of the single sequencer, asked in Appendix E, is answered. It is needed for exactly one thing the semantics cares about: atomicity of competing acts on one commitment, which the typed calculus never performs. For typed programs with statically owned commitments, record production never waits for agreement and replicas converge (Theorem 22.3), and the total order is a choice of presentation. The result is scoped to static ownership: replica ownership transfer and competing acts on one commitment remain agreement problems. The single-sequencer assumption of the kernel is therefore stronger than the calculus needs, and the gap between the two is precisely the gap between the untyped and the typed calculus.

Chapter 23

Eventual Disposition

Progress (Theorem 15.9) is a statement about one configuration: it is never silently stuck. Eventual disposition is a statement about whole executions: every commitment is eventually refused or collapsed. The second does not follow from the first. An execution can make progress forever while neglecting a particular commitment, a transformation can run forever, a verifier can fail to decide, and an outside party can simply never answer. The theorem of this chapter holds only under hypotheses that exclude each of these, and they are stated as named predicates on executions so that it cannot be read as stronger than it is.

23.1 Hypotheses on executions

An *execution* ξ is a finite or infinite sequence of transitions $C_0 \to C_1 \to \cdots$; if finite, it is *maximal* when its last configuration has no transition, discharges included. A *redex* is an occurrence of a top-level prefix, tracked from one configuration to the next while the steps taken are elsewhere.

- $\mathsf{Fair}(\xi)$: no redex is enabled at every configuration from some point on without ever being taken. This is weak fairness, a condition on the scheduler; the calculus itself does not schedule.

- $\mathsf{AllTransformsTerminate}(\xi)$: every transformation redex on a bound commitment in ξ evaluates to a value.

- $\mathsf{AllVerifiersDecide}(\xi)$: every verification redex on a transformed commitment in ξ returns ok or an authenticated fail.

- $\mathsf{AllObligationsResponsive}(\xi)$: every await redex in ξ is eventually settled: discharged, or, for the deadline form, discharged or certified as timed out.

The second and third are quantified over the computations the execution actually invokes, not over the lambda layer as a whole. The lambda layer may be Turing complete; the theorem asks only that the particular transformations and verifiers this execution calls on terminate.

23.2 The theorem

Theorem 23.1 (Eventual disposition)**.** *Let* $\Sigma; \Gamma; \Delta \vdash C_0$ ok *and let* ξ *be an execution from* C_0 *that is maximal or infinite and satisfies* Fair*,* AllTransformsTerminate*,* AllVerifiersDecide*, and* AllObligationsResponsive*. Then every commitment that is live at some configuration of* ξ *is terminal at some later configuration of* ξ*.*

Proof. Let c be live at C_i and suppose, for contradiction, that it is never terminal afterward. By Theorem 15.5, every C_j with $j \geq i$ is well typed and its linear context contains exactly one right for c.

Ownership. Flatten the process of C_j into its top-level parallel components, looking through restriction. By T-Par and Lemma 14.1, exactly one component's context contains the right for c; call it the *owner* O_j. It is not a replication, since T-Rep requires an empty context. By T-Nil it is not $\mathbf{0}$. So O_j is a prefix or a sum of prefixes.

Persistence. If a prefix of O_j is enabled at C_j, it remains enabled until O_j acts. Its premises concern the states of commitments whose rights O_j holds, which no other component can change, since changing a state requires holding its right; authority facts, which are fixed; terminal facts, which are monotone; freshness, which is always satisfiable; and the results of evaluation and verification, which are deterministic. If the prefix is an await, responsiveness supplies its settlement, by discharge or timeout.

Enabledness. By Theorem 15.9 and its proof, every non-await prefix of a well-typed configuration is enabled, given that the computation it invokes terminates, which the second and third hypotheses provide.

Descent. Let $\|P\|$ count the prefix occurrences of P that are not beneath a replication. Each step taken by the owner consumes one prefix, and possibly discards the other summands of a sum, and the right for c passes to a subterm of what remains. Hence the size of the owner of c strictly decreases with each owner step, while steps of other components leave the owner unchanged.

If ξ is infinite, fairness and persistence force the owner to act infinitely often, so its size would decrease without bound, which is impossible. If ξ is finite and maximal, its last configuration has no transition; by responsiveness it is not waiting, since a pending settlement would be a transition; so by Theorem 15.9 it is inert and every commitment is terminal. Either way, c is eventually terminal. □

Remark 23.2 (The well-founded measure)*.* The descent step is where a worry about unbounded waiting is answered, and it is worth making the measure explicit. Suppose a commitment enters verificationFailed and its owner, instead of refusing, waits on an obligation ω_0, then on a fresh ω_1, and so on. Each obligation is eventually discharged, so responsiveness holds, and yet c is never disposed of. This cannot happen in a well-typed program, and the reason is the measure $\|O_j\|$, the number of prefixes in the owner of c that are not beneath a replication. Every await is a prefix of the owner, and every step of the owner, an await's discharge included, removes at least one prefix. The owner cannot replenish itself: the right for c is never under a replication (T-Rep), and the calculus has no other form of recursion. So a commitment can wait on only as many obligations as its owner's text contains, and an infinite succession $\omega_0, \omega_1, \ldots$ for one commitment would need an infinite term. The protocol reading of the same fact is that the session type of a commitment (Section 15.4) is finite and every path through it ends in refuse

or collapse. If the calculus is later extended with recursion that can hold a right, this measure disappears, and "well-founded waiting" becomes a hypothesis of its own.

The proof uses the type system at every step. Ownership is linearity. Persistence depends on ownership: no other component can disable the owner's prefix. Descent depends on T-Rep: rights are never replicated, so the owner is a finite term. And the finite case depends on T-Nil: a program cannot end holding a live right.

23.3 What fails without each hypothesis

Each hypothesis is needed, and each has a smallest counterexecution: a well-typed configuration and an execution satisfying every other hypothesis in which some commitment stays live forever. In each case c has just been opened by a principal a with the authority the program uses.

1. *Unfair scheduling.*

$$!\,\mathsf{pop}\ q'\ \mathsf{as}\ d.\,\mathsf{refuse}_a\ d\ \mathsf{because}\ r.\mathbf{0}\ \mid\ \mathsf{refuse}_a\ c\ \mathsf{because}\ r.\mathbf{0}.$$

   A scheduler that always chooses the replicated component takes a step forever and never the refusal of c, although it is enabled throughout. Every commitment the replication creates is disposed of at once; c never moves.

2. *Divergent transformation.* With $A(c) = \mathsf{bound}(q, b)$ and f a fixed point that diverges on b,

$$\mathsf{transform}\ c\ \mathsf{with}\ f.\,\mathsf{refuse}_a\ c\ \mathsf{because}\ r.\mathbf{0}.$$

   The only redex never completes its evaluation premise, and c stays bound.

3. *Nondeciding verifier.* With $A(c) = \mathsf{transformed}(b, v)$ and a verifier g that on (c, v) neither returns a witness nor issues a certificate,

$$\mathsf{verify}_a\ c\ \mathsf{using}\ g\ \mathsf{then}\ \mathsf{refuse}_a\ c\ \mathsf{because}\ r.\mathbf{0}\ \mathsf{else}\ \mathbf{0}.$$

   Stability (Hypothesis 1.4) is not enough; the verifier must decide.

4. *Permanently unanswered obligation.*

$$\mathsf{await}_e\ c\ \omega.\,\mathsf{refuse}_a\ c\ \mathsf{because}\ r.\mathbf{0},$$

   where e never discharges ω. The configuration is well typed and waiting, not stuck in the sense of Theorem 15.9, and the ledger correctly shows nothing. No other component can refuse c, since the awaiting process holds the right.

5. *An environment that neither discharges nor times out.*

$$\mathsf{await}_e\ c\ \omega.\,P\ \mathsf{within}_b\ t\ \mathsf{else}\ \mathsf{refuse}_a\ c\ \mathsf{because}\ \mathsf{expired}.\mathbf{0},$$

   where e never discharges and the clock authority b never certifies expiry. Declaring a deadline does not by itself end the wait; the deadline is only as responsive as the authority that attests to it.

6. *An unattended verification failure.* With $A(c) = \mathsf{transformed}(b, v)$ and a verifier that returns a certified negative result,

$$\mathsf{verify}_a\ c\ \mathsf{using}\ g\ \mathsf{then}\ P\ \mathsf{else}\ \mathsf{await}_h\ c\ \omega_{\mathrm{review}}.\ \mathsf{refuse}_h\ c\ \mathsf{because}\ r.\mathbf{0},$$

   where the disposition authority h never completes its review. The commitment sits in verificationFailed forever. This is waiting, not a type error, and it exposes the institutional dependency the failed-verification state was introduced to make visible: a negative result that no authorized party acts on.

The last example does not need a hypothesis of its own. In a well-typed program the continuation after a failed verification holds the right, and T-Nil forbids it to stop holding it, so it must either refuse directly, in which case fairness makes the refusal happen, or wait on an obligation. In the second case responsiveness only ends the wait: it guarantees that the await is eventually discharged or timed out, and nothing more. That the refusal after it then happens follows from responsiveness together with the finite descent of the proof of Theorem 23.1, since settling the wait strictly decreases the size of the owner of c, and fairness, which makes that continuation's enabled refusal fire. What some presentations would state separately as "failed-verification obligations receive an authorized response" is, here, the instance of AllObligationsResponsive for the obligation the failed branch awaits. A sixth example shows that typing is needed as well: the untyped process pop q as $c.\mathbf{0}$ satisfies all four hypotheses and leaves c open forever. T-Nil rejects it.

The last two examples are where the liveness question meets institutions. A commitment waiting on a party that never answers is not failing in any sense the calculus can detect; the ledger shows it waiting, correctly. A deadline turns silence into a record, TimedOut, that says the obligation went unanswered within the declared interval. It does not turn silence into refusal. What follows a timeout, whether further waiting, compensation, or refusal, is decided by an authorized rule in the else branch, and choosing that rule is a policy the calculus can express but does not make.

Chapter 24

Bisimulation and the Spectrum as Collapse Rules

The third comparative claim is that rule-indexed collapse changes the theory of equivalence rather than relocating it. This chapter settles it by defining the equivalences explicitly before arranging them, proving inclusions only where they hold, and giving a smallest separating pair for each strict one. The result is not one spectrum but two, one of equivalences on histories and one on processes, and a theorem that says exactly where they meet.

24.1 Equivalences on histories

Each of the following is a collapse rule in the sense of Chapter 2: a lens on replay-defined histories, and the equivalence it induces.

Relation	H_1 and H_2 are related when	Observer
$\approx_{\text{led}}$	they are the same sequence	ledger custodian
$\approx_{\text{tr}}$	they are trace equivalent, $H_1 \sim_I H_2$	auditor of causal history
$\approx_{\text{st}}$	$\text{replay}(H_1) = \text{replay}(H_2)$	state observer
$\approx_{\text{ext}}$	no extension distinguishes them (below)	future experimenter
$\approx_{\text{obs}}$	they have the same observation set O	consumer of observations
$\approx_{\text{lab}}$	same multiset of labels $\langle k, o\rangle$	observer blind to identities
$\approx_{\text{val}}$	same multiset of observed values o	observer blind to rules

Here $H_1 \approx_{\text{ext}} H_2$ holds when, for every sequence of records K, $\text{replay}(H_1K)$ is defined exactly when $\text{replay}(H_2K)$ is, and when both are defined their observation sets agree. It is a testing equivalence in the sense of De Nicola and Hennessy, with ledger extensions as tests and validity together with observation as the outcome.

Proposition 24.1 (The history spectrum)**.**

$$\approx_{\text{led}} \subsetneq \approx_{\text{tr}} \subsetneq \approx_{\text{st}} \subsetneq \approx_{\text{ext}} \subsetneq \approx_{\text{obs}} \subsetneq \approx_{\text{lab}} \subsetneq \approx_{\text{val}} .$$

Proof. The inclusions: equal sequences are trace equivalent; trace equivalent histories replay alike (Proposition 20.10); histories with the same replay state have the same future, since replay is a fold, and the same observations; the empty extension shows $\approx_{\text{ext}} \subseteq \approx_{\text{obs}}$; the last two inclusions forget first commitments and then rules. The separating pairs, each minimal:

- $\approx_{\mathrm{tr}}$ but not $\approx_{\mathrm{led}}$: $\mathsf{Pop}(c_1, q)\,;\mathsf{Pop}(c_2, q)$ and $\mathsf{Pop}(c_2, q)\,;\mathsf{Pop}(c_1, q)$.
- $\approx_{\mathrm{st}}$ but not $\approx_{\mathrm{tr}}$: $\mathsf{Pop}(c, q)\,;\mathsf{Refuse}(a, c, r)$ and $\mathsf{Pop}(c, q)\,;\mathsf{Bind}(a, c, b)\,;\mathsf{Refuse}(a, c, r)$, which both leave c refused for reason r.
- $\approx_{\mathrm{ext}}$ but not $\approx_{\mathrm{st}}$: $\mathsf{Pop}(c, q)\,;\mathsf{Refuse}(a, c, r_1)$ and $\mathsf{Pop}(c, q)\,;\mathsf{Refuse}(a, c, r_2)$. By Theorem 24.4 below, no extension distinguishes them.
- $\approx_{\mathrm{obs}}$ but not $\approx_{\mathrm{ext}}$: the empty history and $\mathsf{Pop}(c, q)$, distinguished by the extension $\mathsf{Bind}(a, c, b)$.
- $\approx_{\mathrm{lab}}$ but not $\approx_{\mathrm{obs}}$: two successful histories that collapse c_1 and c_2 respectively, with the same rule and value.
- $\approx_{\mathrm{val}}$ but not $\approx_{\mathrm{lab}}$: two successful histories collapsing under rules $k_1 \neq k_2$ to the same value (Proposition 21.7). □

24.2 Congruence under extension

A history equivalence $\approx$ is an *extension congruence* when $H_1 \approx H_2$ implies, for every K, that H_1K and H_2K are both replay-defined or both not, and $H_1K \approx H_2K$ when they are.

Proposition 24.2. *$\approx_{\mathrm{led}}$, $\approx_{\mathrm{tr}}$, $\approx_{\mathrm{st}}$, and $\approx_{\mathrm{ext}}$ are extension congruences; $\approx_{\mathrm{obs}}$, $\approx_{\mathrm{lab}}$, and $\approx_{\mathrm{val}}$ are not. Moreover $\approx_{\mathrm{ext}}$ is the coarsest extension congruence contained in $\approx_{\mathrm{obs}}$.*

Proof. Appending K preserves equality and trace equivalence, and replay is a fold, which handles the first three. For $\approx_{\mathrm{ext}}$, extensions of extensions are extensions. The empty history and $\mathsf{Pop}(c, q)$ have the same observations and differ after $\mathsf{Bind}(a, c, b)$, which refutes the last three. If $\approx$ is an extension congruence contained in $\approx_{\mathrm{obs}}$ and $H_1 \approx H_2$, then for every K the extensions are defined together and related by $\approx$, hence have equal observations, so $H_1 \approx_{\mathrm{ext}} H_2$. □

So among the equivalences an observer of outputs might adopt, only those at least as fine as $\approx_{\mathrm{ext}}$ can be used compositionally. An observer who identifies histories by their observations alone will be surprised by the future.

24.3 What the future can see

The coarsest usable equivalence has an exact description. Let α forget, from each lifecycle state, the data that no later record consults:

$$\alpha(\mathsf{open}(q)) = \mathsf{open}, \quad \alpha(\mathsf{bound}(q, b)) = \mathsf{bound}(b), \quad \alpha(\mathsf{transformed}(b, v)) = \mathsf{transformed}(v),$$
$$\alpha(\mathsf{verified}(v, w)) = \mathsf{verified}(v, w), \quad \alpha(\mathsf{verificationFailed}(v, \varphi)) = \mathsf{verificationFailed},$$
$$\alpha(\mathsf{refused}(r)) = \mathsf{refused}, \quad \alpha(\mathsf{collapsed}(o)) = \mathsf{collapsed}(o),$$

keeping the set of settled obligations beside each state. On a capability key, α is the identity: $\alpha(\mathsf{owner}(a)) = \mathsf{owner}(a)$. The current holder of an exclusive capability is consulted by every later exercise and delegation of it, so it cannot be forgotten.

Hypothesis 24.3 (Separation of values)**.** *For every commitment c and value v, verification of (c, v) can decide: some witness is valid for it or some authentic failure certificate exists for it. For values $v \neq v'$, some witness is valid for (c, v) and not for (c, v'), and for every witness valid for both, some rule k has $\mathsf{observe}_k(v) \neq \mathsf{observe}_k(v')$. At least one observation rule exists, every obligation admits an authentic discharge certificate, and, where exclusive authority is in use, every principal can authenticate a delegation of what it owns and at least two principals exist.*

The hypothesis says only that the evidence and observation predicates are rich enough to tell different values apart. Without it, values the future cannot tell apart are, for the future, the same value, and the theorem below holds with α taken modulo that identification.

Theorem 24.4 (Extension full abstraction)**.** *Under Hypothesis 24.3, for replay-defined histories,*

$$H_1 \approx_{\text{ext}} H_2 \quad \Longleftrightarrow \quad \text{dom}(A_1) = \text{dom}(A_2),\ \ \alpha \circ A_1 = \alpha \circ A_2,\ \ O_1 = O_2,$$

where $\langle A_i, O_i \rangle = \text{replay}(H_i)$ is the extended replay, whose map is keyed by commitments and capability keys (Chapter 16). In particular the two histories have the same current owner for every granted capability key.

Proof. ($\Leftarrow$) Each clause of δ is defined or undefined, and produces a state whose α-image is determined, from the α-image of the incoming state and the record alone: binding needs the open kind; transformation needs the bound value b; verification and its failure need the value v and the evidence predicate; refusal needs a live kind; discharge and timeout need a live kind and the settled set; collapse needs v and w, and its observation is $\mathsf{observe}_k(v)$; compensation needs the predecessor's kind; pop needs the name to be absent; a grant needs the capability key absent; a delegation, and an exercise of a granted kind, need the current owner, which α keeps. So by induction on K the two extensions are defined together, keep equal α-images, and add equal observations.

($\Rightarrow$) If $O_1 \neq O_2$ the empty extension distinguishes. Suppose the observations agree but some key differs. Suppose first it is a capability key (c, κ). If it is granted on one side only, a grant of it is valid on exactly one side. If it is granted on both, to owners $a \neq b$, an authenticated delegation from a to a third party, or to b, is valid on exactly one side. Otherwise the key is a commitment c. If c is in one domain only, $\mathsf{Pop}(c, q)$ is valid on exactly one side. If the kinds differ and both are live, the rows of the refusal completion table for distinct live kinds have distinct non-refusal entries, and the corresponding record, which exists by the hypothesis, is valid on exactly one side. If one kind is live and the other terminal, a refusal is valid on exactly one side. Two terminal kinds with equal observations are both refused, since a collapse contributes its triple to O. If the kinds agree and the α-data differ: bound values are distinguished by a transformation record naming one of them; transformed values by a verification with a witness valid for one and not the other; verified states by a collapse carrying one witness, or, if the witnesses agree, by a collapse under a rule that separates the values; settled sets by a discharge of an obligation settled on one side only. □

The theorem's content is in what α forgets. The request that opened a commitment, the input of its transformation once transformed, the certificate of a failed verification, and the reason for a refusal are all invisible to every possible future. So, in general, are the principals who acted: the binder, the refuser, the verifier, and the granter of a capability are recorded in

the history and consulted by nothing after. No continuation of the ledger can depend on them. They are *purely historical*: visible to an auditor reading the past, through $\approx_{\mathrm{tr}}$ or $\approx_{\mathrm{led}}$, and to no one else.

The exception is exact. Historical principals are extension-invisible except where their identity survives in current authority state, and it survives there in one way only: as the current owner of an unconsumed exclusive capability. The smallest separating pair is

$$H_1 = \mathsf{Pop}(c, q)\,;\mathsf{Grant}(g, a, c, \kappa, \sigma), \qquad H_2 = H_1\,;\mathsf{Delegate}(a, b, c, \kappa, \sigma').$$

Their commitment states agree and neither has observations, yet a delegation from a to a third principal d is valid after H_1 and not after H_2, and an exercise of κ on c by b is valid after H_2 and not after H_1. The granter g is invisible in both; the grantee is not. Past attribution may be observationally inert; current authority never is. In the fragment without exclusive capabilities, where authority is a fixed policy, no principal's identity survives in the state and every principal is purely historical.

24.4 Equivalences on processes

The classical spectrum is a spectrum of equivalences on processes, all of which compare possible futures: trace equivalence, failures equivalence, the testing equivalences, ready simulation, bisimilarity, and the rest of van Glabbeek's linear-time/branching-time spectrum. Through the erasure of Chapter 10, each applies to disposition processes as it applies to CCS terms, and nothing about the disposition calculus changes that spectrum.

The two spectra are independent.

Proposition 24.5 (Past and future are orthogonal)**.** *There are configurations with identical histories whose processes are not trace equivalent, and configurations whose processes are strongly bisimilar and whose histories are related by none of the history equivalences finer than* $\approx_{\mathrm{val}}$.

Proof. For the first, take the same history with processes $\mathbf{0}$ and $\mathsf{pop}\ q\ \mathsf{as}\ c.\mathbf{0}$. For the second, take process $\mathbf{0}$ with two histories that collapse the same commitment under different rules to the same value. □

An observer's notion of "the same computation" is therefore a pair: an equivalence on pasts and an equivalence on futures. A ledger auditor, a state observer, and a process observer are different points in this product and need not identify the same executions.

24.5 Outcome

The third comparative claim is settled, in a precise form. The classical spectrum is untouched: it is a spectrum of equivalences on futures, and rule-indexed collapse does not add to it. What rule-indexed collapse adds is a second axis, a spectrum of equivalences on pasts (Proposition 24.1), orthogonal to the first (Proposition 24.5). The two meet at one point: $\approx_{\mathrm{ext}}$, the coarsest past-equivalence compatible with extension, is exactly equality of the future-relevant part of the state (Theorem 24.4), and it is strictly coarser than state equality. Each individual lens is an ordinary equivalence relation, and none of the mathematics here is new. What is

new is the arrangement: the insistence that equivalence of computations be indexed by which of these observers is asking, and the theorem that shows the reasons for decisions are visible to exactly one of them.

Chapter 25

Checking the Layer Translation

Chapter 3 proved forward simulation and backward adequacy for the translation of disposition records into kernel histories. This chapter checks the translation against the remaining criteria of Chapter 4 and states precisely the one question it leaves open.

25.1 The criteria

The translation is a translation of ledgers, not of processes, so the criteria are read with ledgers as terms, appending a record as the only operator, and replay-definedness as the admissibility of a term.

Compositionality, in the form appropriate to histories.
The translation of $H \cdot \rho$ is $[\![H]\!]$ followed by $[\![\rho]\!]_H$, and $[\![\rho]\!]_H$ depends on H only through the last record of each commitment involved and the set of options of the subject not yet popped. It is therefore not compositional in Gorla's sense, which asks for one fixed context per operator, but it is a *transducer*: a fixed kernel program for each record kind, reading the current state of the kernel histories it touches. For translations of histories, whose meaning is a fold, this is the natural form of compositionality, and the book claims no more.

Name invariance.
For every permutation σ of commitment names, $[\![\sigma H]\!] = \sigma[\![H]\!]$, since options are records and permutations act on records pointwise, and the guards are equivariant because they are the premises of rules that are (Theorem 5.3).

Operational correspondence.
Forward simulation and backward adequacy (Theorems 3.5 and 3.7).

Divergence reflection.
Each record translates to at most three kernel events, so a translated history of finite length has finite translation, and under the opaque reading nothing in the kernel diverges on behalf of a record.

Success sensitiveness.
With success a collapse under a designated rule, a ledger contains a success record exactly when its translation pops a committed observation under that rule.

Injectivity.
Decoding a translation recovers the ledger, so the translation is injective and fully abstract for $\approx_{\mathrm{led}}$ on the disposition side and equality on the kernel side. Full abstraction for coarser equivalences would need a kernel analogue of trace equivalence, which the kernel's own specification would have to supply.

Under the opaque reading and at the level of ledgers, the translation therefore meets every criterion, compositionality in the transducer form. The disposition calculus is a conservative guarded presentation of the kernel in the sense Chapter 3 claimed.

25.2 The transparent reading

What the criteria do not show is that the kernel performs the computations the guards check. Under the opaque reading, the value of a transformation enters as recorded data whose correctness the guard asserts; the kernel commits to it without computing it. The transparent reading would have the kernel compute it.

Hypothesis 25.1 (Transparent reading, conjectured)**.** *Let* Enc *be the kernel's encoding of lambda terms, with abstraction as* $\mathsf{Bind}(x, \Omega)$ *and application as* $\mathsf{Collapse}_{\mathrm{subst}}(\mathsf{Bind}(u, x))$*. For closed terms* f*,* b *and every value* v*,*

$$\mathsf{Collapse}^{*}_{\mathrm{subst}}\big(\mathsf{Bind}(\mathrm{Enc}\, f, \mathrm{Enc}\, b)\big) = \mathrm{Enc}\, v \quad \Longleftrightarrow \quad f\, b \Downarrow v,$$

where $\mathsf{Collapse}^{*}_{\mathrm{subst}}$ *iterates substitution collapse to a normal form, and the left side is undefined exactly when* $f\, b$ *diverges.*

The canonical specification states a theorem relating its substitution collapse to a single β-step. The conjecture asks for more in three respects: that iterating the collapse follows the call-by-value strategy of Chapter 7 rather than some other; that it agrees with evaluation on products, sums, and fixed points, not only on pure abstraction and application; and that it diverges exactly when evaluation does. A proof would turn the conservative guarded presentation into a formalization in the full sense, with the kernel computing what the guard now asserts. A failure would not disturb any ledger theorem of this book. It would locate precisely the computational content the kernel does not supply, and would make the lambda layer a genuine addition to the kernel rather than a presentation of part of it.

Chapter 26

Typed Disposition Safety

All the pieces are in place. This chapter states the book's central safety theorem and proves it by a short induction, as the proof spine promised, because each clause has already been established by a local result.

Theorem 26.1 (Typed disposition safety)**.** *Let* $\Sigma_0; \Gamma_0; \Delta_0 \vdash C_0$ ok*, and let* $C_0 \to C_1 \to \cdots \to C_n$ *be any finite execution. Then there are* Σ_n*,* Γ_n*,* Δ_n *such that:*

(1) $\Sigma_n; \Gamma_n; \Delta_n \vdash C_n$ ok*;*

(2) every live commitment of C_n *has exactly one right in* Δ_n*, and every terminal commitment has none;*

(3) every attributed transition of the execution was authorized;

(4) every verified state and every Verify *record carries a witness valid for its own commitment and value;*

(5) every collapsed commitment has a complete ledger path through opening, binding, transformation, verification, and collapse;

(6) every refusal appended during the execution is present in H_n*;*

(7) no commitment has two terminal records in H_n*;*

(8) $\mathrm{replay}(H_n) = \langle A_n, O_n \rangle$*;*

(9) the process of C_n *contains no well-typed continuation acting on a commitment whose right its context does not hold, and at most one component holds each right.*

Proof. By induction on n, the case $n = 0$ being the hypothesis. For the step, Theorem 15.5 carries (1) from C_n to C_{n+1}. The remaining clauses are properties of any well-typed, consistent configuration and of the execution leading to it:

- (2) is condition (C3) of (1), that is, the corollary to Theorem 15.5;
- (3) is Theorem 9.9, applied to each step;
- (4) is Corollary 18.3 with Lemma 14.7, and non-transferability under Hypothesis 2.10;

- (5) is Theorem 18.4;
- (6) is Theorem 18.1 with history monotonicity;
- (7) is Corollary 18.5;
- (8) is condition (C2) of (1), maintained by Proposition 20.3;
- (9) is Lemma 15.1, since each prefix acting on c requires c's right in its context, together with Lemma 14.1, since two components cannot both hold it.

The untyped results used here were proved for executions from a consistent configuration, which (1) guarantees. □

Two things are deliberately absent from the theorem. It says nothing about whether commitments are eventually disposed of; that is Theorem 23.1, and it needs fairness, terminating computations, deciding verifiers, and responsive obligations, none of which a safety theorem should assume. And it says nothing about whether the ledger is complete; that is Proposition 20.6, and it needs a checkpoint retained outside the history.

The three validation levels of the Preface can now be seen together. The laboratory's bounded propositions check the untyped clauses (5), (6), and (7) and the replay clause (8) on every ledger of a finite class, and found the specification errors the book corrected along the way. The untyped proofs establish (3) through (8) for every execution of every program. The typed metatheory adds (1), (2), and (9), which the untyped calculus cannot even state: that rights are neither duplicated nor abandoned, and that no well-typed program contains a competitor for a commitment it does not own.

Chapter 27

A Map of Positions

This chapter collects the comparisons of the book in one place and answers the three claims of the Preface in their final form.

27.1 The comparisons

Comparison	Result	Outcome	Grade
Kernel (ledger level)	Thms. 3.5, 3.7; Ch. 25	conservative guarded presentation, opaque reading	proved
Kernel computes	Hyp. 25.1	transparent reading	conjecture
Lambda layer	Lem. 7.2; normalization	deterministic; terminating without fix	proved; cited
Linear logic	Props. 8.1, 8.2	kernel affine, rights linear, facts exponential	proved; cited
CCS into typed calculus	Thm. 10.5	separation: no compositional, success-sensitive encoding	proved
Communication-free CCS	App. D	all criteria, fully abstract (weak bisimilarity)	proved
Typed calculus to CCS	Props. 10.1, 10.2	erasure complete, not sound, not fully abstract	proved
CSP failures model	Prop. 11.1, Thm. 11.2	refusal notions independent; record invisible without instrumentation	proved
π-calculus	Thm. 12.2, Cor. 12.3	strict inclusion: typed calculus encodes into π, not conversely	proved
π-calculus, integrity	Prop. 12.4	scope versus authority: π loses invariants on extrusion	proved
Choice hierarchy	Prop. 13.1, Cor. 13.3	separate choice; symmetry broken by the arbiter, untyped only	proved
Event structures	Thm. 21.5, Prop. 21.7	ledger semantics is a labelled prime event structure; residue η	proved
Replicated data types	Thm. 22.3	coordination free under static ownership	proved
Equivalence spectrum	Prop. 24.1, Thm. 24.4, Prop. 24.5	second, orthogonal spectrum on pasts; meets futures at $\approx_{\text{ext}}$	proved (Hyp. 24.3)

27.2 The three claims

First claim: event structures are the nearest relative. Settled positively, with a residue. At the ledger level the semantics of a configuration is a labelled prime event structure whose configurations are its executions up to trace equivalence, and consistency in it is binary (Theorem 21.5). The residue has two parts. The ordinary reduct forgets the annotations on which authority, evidence, and observation provenance depend (Proposition 21.7), and the calculus supplies something no event structure contains: the rule that generates the structure from evidence and authority predicates over unbounded data.

Second claim: recorded refusal has no classical counterpart. Settled in its weak form only. The failures model of CSP does not represent a recorded refusal without an added event, and the CSP and disposition notions of refusal are independent, neither implying the other (Theorem 11.2, Proposition 11.1). The stronger reading, that no formalism can carry a recorded refusal, is false and was never claimed: with a rich enough alphabet any trace-based formalism can carry it as an event.

Third claim: rule-indexed collapse changes the theory of equivalence. Settled in a precise form. The classical spectrum, a spectrum of equivalences on futures, is unchanged. Rule-indexed collapse adds a second spectrum, of equivalences on pasts, orthogonal to the first (Propositions 24.1 and 24.5). The two meet at $\approx_{\text{ext}}$, the coarsest past-equivalence compatible with extension, which is strictly coarser than equality of state: reasons, requests, failure certificates, and past principals are visible to no future (Theorem 24.4), except that the current holder of an unconsumed exclusive capability is part of the state and is visible to every future. Past attribution may be observationally inert; current authority never is.

27.3 The position

In expressive power the typed disposition calculus is modest. It is a sub-calculus of the π-calculus with data and separate choice up to an encoding meeting every criterion, and it cannot encode synchronization, because its parallel components do not interact (Lemma 10.3). Its semantics is a labelled prime event structure. Nothing it computes lies outside the lambda calculus, and nothing it communicates lies outside CCS.

What it adds lies elsewhere, and the comparisons locate it. Refusal is recorded, attributed, and reasoned, as an object distinct from a CSP refusal set. Integrity rests on authority and replay rather than on the secrecy of names, so it survives the scope extrusion under which the π-encoding fails. Verification classifies evidence without disposing of anything, and every disposition traces to a principal entitled to make it. Equivalence is indexed by the observer, and the reasons for decisions are visible to exactly one kind of observer, the one who reads the past. Under static ownership, typed programs converge without consensus, because they never contend. These are guarantees about every execution rather than capabilities some programs can exercise, and they are what the book set out to establish.

What rests on what. Proved in the book: every result in the table marked "proved". Cited without re-derivation, and load-bearing where used: Gorla's encodability framework; Palamidessi's separation result; strong normalization of the simply typed calculus; balance of multiplicative linear logic; correctness of bracket abstraction; the failures semantics of CSP; Herlihy's hierarchy; and the security of signatures and hash functions behind certificates and anchors. Conjectural: the transparent reading, the untyped calculus's encoding of asynchronous single-use CCS, and the sound translation of the typed calculus with finite data into CCS, which is sketched but not carried out.

Appendix A

Proof Spine

The proof dependency is: unique abstract parsing and alpha-invariance; fresh commitment introduction and lifecycle preservation; append-only history; value and process substitution; subject reduction; authority preservation and binding integrity; witness ancestry; collapse soundness and terminal exclusivity; replay correspondence; compensation conservation; and finally typed disposition safety. Liveness is stated separately because it requires fairness, responsive external dependencies, terminating transformations, and decidable verification. Safety holds for every finite execution without those assumptions.

Result	Status	Location
Unique tokenization	proved	Lem. 6.1
Unique parsing	proved (LL(1))	Thm. 6.2
Printer adequacy	proved	Thm. 6.3
Equivariance	proved	Ch. 5
Commitment identity	proved	Thm. 9.5
Lifecycle and history monotonicity	proved	Ch. 9
Authorized transition	proved (static policy)	Thm. 9.9
Source authenticity of verification failures	proved	Prop. 9.10
Ledger acceptance of verification failures	proved (given certificates)	Prop. 9.11
Diamond for independent steps	proved	Prop. 9.3
Ledger path lemma	proved	Lem. 17.1
Refusal preservation	proved	Thm. 18.1
Binding integrity	proved (exclusivity: typed)	Ch. 18
Witness ancestry	proved	Cor. 18.3
Collapse soundness	proved	Thm. 18.4
Terminal exclusivity	proved	Cor. 18.5
Compensating continuation	proved	Thm. 19.1
Compensation ancestry	proved	Lem. 19.4
Consistency invariant	proved	Prop. 20.3
Replay correspondence	proved	Thm. 20.1
Prefix indistinguishability	proved	Thm. 20.4
Subhistory indistinguishability	proved	Thm. 20.5
Relative completeness	proved (conditional)	Prop. 20.6
Idempotent disposition	proved (under coherence)	Ch. 20
Replay respects commutation	proved	Prop. 20.10

Result	Status	Location
Causal shuffle	proved	Thm. 2.9
Modal terminal exclusivity	proved	Prop. 2.4
No cycles, one re-entry	proved	Prop. 2.12
Kernel translation: forward simulation	proved (ledger level)	Thm. 3.5
Kernel translation: backward adequacy	proved (ledger, universal process)	Thm. 3.7
Kernel translation: remaining criteria	proved (transducer form)	Ch. 25
Kernel translation: transparent reading	conjecture	Hyp. 25.1
Determinism of evaluation	proved	Lem. 7.2
Evaluation or divergence (no stuck terms)	proved	Lem. 7.1
Kernel affine, no linear discarding	proved; cited	Props. 8.1, 8.2
Occurrence structures trace invariant	proved	Thm. 21.2
Binary consistency of $\mathcal{W}(C)$ (extended replay)	proved	Thm. 21.5
Forgetting annotations	proved	Prop. 21.7
CCS erasure: forward, not sound	proved	Props. 10.1, 10.2
Pruning (typed non-interference)	proved	Lem. 10.3
Context decomposition	proved	Cor. 10.4
No encoding of CCS synchronization	proved	Thm. 10.5
CSP: refusal notions independent	proved	Prop. 11.1, Thm. 11.2
Encoding into the π-calculus	proved	Thm. 12.2, Cor. 12.3
Scope versus authority	proved	Prop. 12.4
Election by race; typed cannot race	proved	Prop. 13.1, Cor. 13.3
Losers detect the loss (test-and-set)	proved	Prop. 13.2
Coordination-free convergence (static ownership)	proved	Thm. 22.3
History spectrum and congruence	proved	Props. 24.1, 24.2
Extension full abstraction, with capability keys	proved (Hyp. 24.3)	Thm. 24.4
Communication-free CCS, fully abstract	proved	Prop. D.1
Value substitution, linear and unrestricted	proved	Ch. 14
Canonical forms for rights	proved	Lem. 14.7
Preservation	proved	Thm. 15.5
Progress with typed waiting	proved	Thm. 15.9
Protocol fidelity (single commitment)	proved	Thm. 15.10
Typed disposition safety	proved	Thm. 26.1
Eventual disposition	proved (four hypotheses)	Thm. 23.1
Non-derivability of authority	proved (static policy)	Prop. 16.5
Delegation authenticity	proved	Prop. 16.1
Conservation of exclusive capabilities	proved	Prop. 16.2
Preservation under transfer	proved	Prop. 16.3
Delegation chain soundness	proved	Prop. 16.4

Appendix B

Complete Syntax, Typing, and Transition Rules

This appendix collects the definitions of the calculus in one place, in tabular form, with a pointer to where each is stated and discussed.

B.1 Syntax

The concrete grammar is in Chapter 6; the abstract grammar of processes, expressions, and types is in Chapters 5, 7, and 14. Lifecycle states:

$\mathsf{open}(q)$, $\mathsf{bound}(q, b)$, $\mathsf{transformed}(b, v)$, $\mathsf{verified}(v, w)$, $\mathsf{verificationFailed}(v, \varphi)$ (live),
$\mathsf{refused}(r)$, $\mathsf{collapsed}(o)$ (terminal).

B.2 Prefixes, records, and authority

Prefix	Record appended	From state	Authority
$\mathsf{pop}\ q\ \mathsf{as}\ c$	$\mathsf{Pop}(c, q)$	c fresh	none
$\mathsf{pop}_a\ q\ \mathsf{as}\ c\ \mathsf{after}\ d$	$\mathsf{Compensates}(a, c, q, d)$	c fresh, d terminal	compensate on d
$\mathsf{bind}_a\ c\ \mathsf{to}\ b$	$\mathsf{Bind}(a, c, b)$	open	bind
$\mathsf{transform}\ c\ \mathsf{with}\ f$	$\mathsf{Transform}(c, f, b, v)$	bound, $f\, b \Downarrow v$	none
$\mathsf{verify}_a \ldots$, success	$\mathsf{Verify}(a, c, v, w)$	transformed, $g \Downarrow \mathsf{ok}(w)$	verify
$\mathsf{verify}_a \ldots$, failure	$\mathsf{VerificationFailed}(a, c, v, \varphi)$	transformed, $g \Downarrow \mathsf{fail}(\varphi)$	verify
$\mathsf{refuse}_a\ c\ \mathsf{because}\ r$	$\mathsf{Refuse}(a, c, r)$	any live	refuse
$\mathsf{collapse}_{a,k}\ c$	$\mathsf{Collapse}(a, c, k, w, o)$	verified, $o = \mathsf{observe}_k(v)$	collapse
$\mathsf{await}_a\ c\ \omega$	$\mathsf{Discharged}(a, c, \omega, \varphi)$	live, certificate	discharge on ω
$\ldots\,\mathsf{within}_b\ t$, expiry	$\mathsf{TimedOut}(b, c, \omega, t, \varphi)$	live, certificate	clock on ω
$\mathsf{delegate}_{a \to b}\langle c, \kappa \rangle$	$\mathsf{Delegate}(a, b, c, \kappa, \sigma)$	a owns (c, κ)	ownership

Every rule is in Chapter 9, except delegation, in Chapter 16. Each appends exactly the record shown and updates the commitment map at its subject by the corresponding clause of

δ (Definition 1.3); collapse also adds (c, k, o) to the observation set, and delegation updates the ownership map. The contextual rules for parallel composition, choice, restriction, and structural congruence are in Chapter 9.

B.3 Replay clauses

Replay (Chapter 20) applies each record to the reconstructed state by the clause of δ for its kind, which is the table above read from the "From state" column, with the evaluation and verification premises replaced by the checks that can be made from the record alone: for verification, valid(w, c, v); for its failure, discharge, and timeout, checkCert on the certificate carried; for collapse, $o = \mathsf{observe}_k(v)$ and the witness held in the state; for compensation, terminality of the predecessor; for delegation, current ownership and the signature. Transformation is checked only for agreement of b with the bound state: the result v is taken from the record, which is the opaque reading of Chapter 3.

B.4 Typing

Rule	Consumes	Produces for the continuation
T-Pop	nothing (fresh c added to Σ)	Commit$\langle c, A, \mathsf{Open}\rangle$
T-Comp	fact Terminal$\langle d, t\rangle$ consulted	Commit$\langle c, A, \mathsf{Open}\rangle$
T-Bind	Commit$\langle c, A, \mathsf{Open}\rangle$	Commit$\langle c, A, \mathsf{Bound}\rangle$
T-Trans	Commit$\langle c, A, \mathsf{Bound}\rangle$	Commit$\langle c, B, \mathsf{Transformed}\rangle$
T-Ver, success	Commit$\langle c, B, \mathsf{Transformed}\rangle$	Commit$\langle c, B, \mathsf{Verified}\rangle$
T-Ver, failure	Commit$\langle c, B, \mathsf{Transformed}\rangle$	Commit$\langle c, B, \mathsf{VerificationFailed}\rangle$, fact
T-Ref	Commit$\langle c, A, s\rangle$, s live	token Refusal$\langle c, A, s\rangle$, terminal fact
T-Col	Commit$\langle c, B, \mathsf{Verified}\rangle$	token Observed$\langle c, B, k\rangle$, terminal fact
T-Await, T-Deadline	nothing	the same right, in each branch
delegation	Cap$\langle a, c, \kappa\rangle$	Cap$\langle b, c, \kappa\rangle$
T-Nil	tokens only; no rights	
T-Par	$\Delta_1 \uplus \Delta_2$ split	each side its part
T-Sum	Δ	Δ in each summand
T-Res, T-Rep	Δ; empty for replication	unchanged

Every attributed prefix also requires its authority fact in Γ, or, for exclusive authority, the capability in Δ. The full rules are in Chapters 14 and 16; the configuration invariant tying them to the runtime is Definition 14.6.

Appendix C

Encodability Criteria, Stated Formally

This appendix collects the notation of Chapter 4.

Calculi. A calculus is a tuple $\mathcal{L} = \langle \mathcal{T}, \mathcal{N}, \mathcal{A}, \to, \Downarrow \rangle$ of terms, names, labels with a distinguished $\tau \in \mathcal{A}$, a labelled transition relation $\to \subseteq \mathcal{T} \times \mathcal{A} \times \mathcal{T}$, and a success predicate $\Downarrow \subseteq \mathcal{T}$. Weak transitions: $\Rightarrow = (\xrightarrow{\tau})^*$ and $\xRightarrow{\mu} = \Rightarrow \xrightarrow{\mu} \Rightarrow$ for $\mu \neq \tau$, $\xRightarrow{\tau} = \Rightarrow$.

Translations. A translation from $\mathcal{L}_s$ to $\mathcal{L}_t$ is a triple $\langle [\![\cdot]\!], \varphi, \widehat{\cdot} \rangle$ with $[\![\cdot]\!] : \mathcal{T}_s \to \mathcal{T}_t$, a renaming policy $\varphi : \mathcal{N}_s \to \mathcal{N}_t^+$ injective on distinct names with pairwise disjoint images, and an action translation $\widehat{\cdot} : \mathcal{A}_s \to \mathcal{A}_t$ with $\widehat{\tau} = \tau$. It is equipped with an equivalence $\asymp \subseteq \mathcal{T}_t \times \mathcal{T}_t$.

Criteria.

(E1) *Compositionality*: for every k-ary operator op and finite $N \subseteq \mathcal{N}_s$ there is C_{op}^N such that for all $S_1, \ldots, S_k$ with $\bigcup \mathrm{fn}(S_i) = N$, $[\![\mathrm{op}(S_1, \ldots, S_k)]\!] = C_{\mathrm{op}}^N([\![S_1]\!], \ldots, [\![S_k]\!])$.

(E2) *Name invariance*: for every injective $\sigma : \mathcal{N}_s \to \mathcal{N}_s$ there is σ' with $\varphi(\sigma(n)) = \sigma'(\varphi(n))$ for all n and $[\![\sigma S]\!] = \sigma' [\![S]\!]$.

(E3) *Completeness*: $S \xRightarrow{\mu} S'$ implies $[\![S]\!] \xRightarrow{\widehat{\mu}} T \asymp [\![S']\!]$ for some T.

(E4) *Soundness*: $[\![S]\!] \xRightarrow{\nu} T$ implies there are μ, S', T' with $\widehat{\mu} = \nu$, $S \xRightarrow{\mu} S'$, $T \Rightarrow T'$, and $T' \asymp [\![S']\!]$.

(E5) *Divergence reflection*: if $[\![S]\!]$ has an infinite τ-sequence, so has S.

(E6) *Success sensitiveness*: $S \Downarrow \iff [\![S]\!] \Downarrow$.

(E7) *Full abstraction* with respect to $\approx_s, \approx_t$: $S_1 \approx_s S_2 \iff [\![S_1]\!] \approx_t [\![S_2]\!]$.

Homomorphism. A translation is homomorphic on op when $C_{\mathrm{op}}^N = \mathrm{op}$ for every N.

Separation. $\mathcal{L}_s$ is separated from $\mathcal{L}_t$ with respect to a set of criteria when no translation satisfies all of them. The standard source of such results is Palamidessi's separation of mixed-choice from asynchronous π-calculus; the book's separations (Chapters 10 and 11) are stated relative to the criteria they name, and the criteria are the load-bearing part of each statement.

Appendix D

A Worked Encoding: A CCS Fragment into the Calculus

Theorem 10.5 rules out a compositional encoding of CCS synchronization into the typed calculus. This appendix gives the encoding that remains possible once synchronization is set aside, and checks each criterion of Appendix C explicitly.

D.1 The fragment

Let CCS_0 be the communication-free fragment:

$$S ::= \mathbf{0} \mid a.S \mid S + S \mid S \mid S,$$

over action names a, with a designated action $\checkmark$, and the labelled transitions

$$a.S \xrightarrow{a} S, \qquad \frac{S \xrightarrow{a} S'}{S + T \xrightarrow{a} S'}, \qquad \frac{S \xrightarrow{a} S'}{S \mid T \xrightarrow{a} S' \mid T},$$

with the symmetric rules for $+$ and $|$. No action has a complement here, so there are no internal steps, and $S \Downarrow$ means that S can perform $\checkmark$ after some sequence of actions. Restriction is left out: without communication its only effect is to block actions of the restricted names, which a compositional translation cannot do without inspecting its argument.

D.2 The translation

Fix a principal $\star$ with a uniform policy granting it refusal authority over every commitment. Define

$$\begin{aligned} [\![\mathbf{0}]\!] &= \mathbf{0}, & [\![a.S]\!] &= \mathsf{pop}\ a\ \mathsf{as}\ c.\,\mathsf{refuse}_\star\ c\ \mathsf{because}\ \texttt{"done"}.\,[\![S]\!], \\ [\![S + T]\!] &= [\![S]\!] + [\![T]\!], & [\![S \mid T]\!] &= [\![S]\!] \mid [\![T]\!], \end{aligned}$$

with c fresh for $[\![S]\!]$. An action becomes a commitment opened with the action's name as its request and then refused, so that the ledger records each action and leaves no commitment live. The action translation sends a to the label $\mathsf{pop}(\cdot, a)$, ignoring the fresh name, and treats

refusal labels as internal. Success in the target is a Pop record with request ✓. For $\asymp$ take equality of processes, ignoring the refused commitments and the tokens the translation leaves behind.

Every image is well typed from the empty configuration. The pop gives the open right, the refusal consumes it (T-Ref, with the uniform authority fact), and the continuation holds only tokens, which T-Nil, T-Sum, and T-Par accept. Every summand of an image begins with a prefix, so the images are in the guarded fragment of Chapter 12 as well.

D.3 The criteria

(E1) Compositionality.
Homomorphic on $\mathbf{0}$, $+$, and $|$; for prefixing the context is

$$\mathsf{pop}\ a\ \mathsf{as}\ c.\ \mathsf{refuse}_\star\ c\ \mathsf{because}\ \texttt{"done"}.[\cdot].$$

(E2) Name invariance.
Action names become requests, which are values. A renaming σ of action names is matched by the same renaming of requests, and $[\![\sigma S]\!] = \sigma[\![S]\!]$ by induction on S.

(E3) Completeness.
By induction on the derivation of $S \xrightarrow{a} S'$: the prefix case fires the pop and then the refusal, reaching $[\![S']\!]$; the sum and parallel cases lift through the corresponding rules of Chapter 9.

(E4) Soundness.
A visible target step is a pop in some summand path of the image, which corresponds to an initial action of S; the refusal that follows it is enabled and reaches the image of the corresponding derivative. Pending refusals in other components commute with everything, so they are absorbed by $\Rightarrow$.

(E5) Divergence reflection.
Every internal step of an image is a refusal, and each refusal consumes a right created by a preceding pop. There are never more internal steps than visible ones, so there is no infinite internal sequence.

(E6) Success sensitiveness.
S can perform ✓ exactly when its image can append $\mathsf{Pop}(c, ✓)$, by (E3) and (E4).

Proposition D.1 (Full abstraction)**.** *For* CCS_0 *terms,* S_1 *and* S_2 *are strongly bisimilar if and only if their images are weakly bisimilar, with visible labels* $\mathsf{pop}(\cdot, a)$ *compared by their requests and refusals internal.*

Proof. Say a target process is a *pending image* of S if it is $[\![S]\!]$ with some components preceded by a pending refusal prefix. A pending refusal is an internal step that is always enabled, commutes with every other step, and cannot be preempted, so each pending image is weakly bisimilar to $[\![S]\!]$. From left to right, if $\mathcal{R}$ is a strong bisimulation on CCS_0, then relating pending images of S_1 and S_2 for each $(S_1, S_2) \in \mathcal{R}$ is a weak bisimulation, by (E3) and (E4). From right to left, the source has no internal steps, so a weak bisimulation between images, restricted to images of source terms and projected back, matches each visible step of one source term by a single visible step of the other, which is a strong bisimulation. □

D.4 What the example shows

The encoding meets every criterion and is fully abstract, and it is nearly trivial, which is the point. Once synchronization is removed, CCS is interleaving of independent sequential agents, and the typed calculus expresses exactly that. What the translation adds is not behaviour but record: each action leaves a pop and a refusal in the ledger, attributed and replayable. The appendix is the positive half of the CCS comparison; Theorem 10.5 is the negative half, and between them they locate the boundary at synchronization precisely.

Appendix E

Open Problems

Questions raised in earlier drafts and answered in this one are listed at the end, with the result that answered each.

1. *The transparent reading.* Whether the kernel's encoding of application agrees with call-by-value evaluation in the lambda layer, including on divergence (Hypothesis 25.1, Chapter 25).

2. *Asynchronous CCS in the untyped calculus.* In the untyped calculus, one component's act can enable another's through the shared commitment map, a one-shot, one-way signal (Chapter 10). Does it suffice to encode asynchronous CCS with single-use channels, and what does the encoding cost in the criteria?

3. *The finite-data translation into CCS.* With finitely many values each commitment is a finite-state agent, which suggests a sound translation of the typed calculus into CCS (Chapter 10). It is sketched, not proved.

4. *A hardened π-encoding.* Cells that check authority and a log that carries signatures would protect the π-encoding against contexts that learn commitment names (Proposition 12.4). A proof of correctness for such a protocol against arbitrary contexts is open.

5. *Verifier versioning.* The theorems of Part V assume Hypothesis 1.4. With versioned verifiers, the version must be recorded and witness ancestry restated relative to it; whether an old verification remains evidence under a new version is a policy question the calculus should make explicit rather than decide.

6. *Competing compensations.* Two authorized principals may compensate one terminal commitment, and both successors are admitted; no consistency question arises (Chapter 22). Whether a policy should forbid this, or a later act arbitrate between successors, is open.

7. *Mutable store.* Replay correspondence excludes S. The exact recording obligations under which S becomes reconstructible remain to be stated.

8. *Denied attempts.* An unauthorized attempt has no transition and leaves no record. Recording DeniedAttempt on the annotation channel would make attempted overreach auditable without making it causal; who records it, and whether an untrusted component can flood that channel, is open.

9. *Moving ownership and revocation.* Failover that transfers a commitment to a new owner, and revocation of a delegated capability, both require agreement on an order between competing acts (Chapters 22 and 16). Neither is in the core calculus.

10. *Recursion that holds a right.* Eventual disposition rests on a well-founded measure that disappears if a right can be held under recursion (Chapter 23). An extension with such recursion would need well-founded waiting as a hypothesis or a proof.

11. *Encoding delegation into the π-calculus.* The encoding of Chapter 12 omits exclusive capabilities. Adding a cell per capability key, read by exercises and written by grants and delegations, is the evident extension; checking it against the criteria is open.

12. *Consensus with readable state.* The atomic commitment map is a test-and-set object, but processes cannot read the map, so consensus on values is not solvable by the race alone (Chapter 13). What a read primitive on replay state would add, and at what cost to pruning, is open.

Answered in this edition.

- *Verification authority and refusal*: resolved by the verificationFailed state (Chapter 16).
- *Recording the road not taken*: more informative, not more expressive (Chapter 13).
- *Restriction and visibility*: restriction is lexical scope; integrity rests on authority, not secrecy (Proposition 12.4).
- *Anchoring the ledger's end*: relative completeness from a signed checkpoint retained independently (Proposition 20.6), per replica if need be (Chapter 22).
- *The purpose of total order*: atomicity of competing acts on one commitment, which typed programs never perform, and replica ownership transfer, which static ownership excludes (Theorem 22.3).
- *Principals and the future*: past principals are extension-invisible; the current owner of an exclusive capability is not (Theorem 24.4).

Bibliography

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM* 31(3), 1984.

[3] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. *POPL*, 1977.

[5] J.-Y. Girard. Linear logic. *Theoretical Computer Science* 50, 1987.

[6] D. Gorla. Towards a unified approach to encodability and separation results for process calculi. *Information and Computation* 208(9), 2010.

[7] J. M. Hellerstein and P. Alvaro. Keeping CALM: when distributed consistency is easy. *Communications of the ACM* 63(9), 2020.

[8] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* 13(1), 1991.

[9] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. *IJCAI*, 1973.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[11] J. R. Hindley and J. P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, 2008.

[12] K. Honda. Types for dyadic interaction. *CONCUR*, 1993.

[13] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21(7), 1978.

[14] A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Report PB-78, Aarhus University, 1977.

[15] R. Milner. *A Calculus of Communicating Systems*. LNCS 92, Springer, 1980.

[16] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation* 100(1), 1992.

[17] U. Montanari and F. Rossi. Contextual nets. *Acta Informatica* 32(6), 1995.

[18] P. Baldan, A. Corradini, and U. Montanari. Contextual Petri nets, asymmetric event structures, and processes. *Information and Computation* 171(1), 2001.

[19] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science* 13, 1981.

[20] C. Palamidessi. Comparing the expressive power of the synchronous and asynchronous π-calculi. *Mathematical Structures in Computer Science* 13(5), 2003.

[21] D. Park. Concurrency and automata on infinite sequences. *Theoretical Computer Science*, LNCS 104, 1981.

[22] C. A. Petri. *Kommunikation mit Automaten*. Dissertation, University of Bonn, 1962.

[23] G. D. Plotkin. A structural approach to operational semantics. DAIMI FN-19, Aarhus University, 1981.

[24] F. S. Fatemi Nasrollahi, E. Newby, and R. Albert. Modularity-based approach identifies small sets of control nodes in synthetic and biological Boolean networks. *Scientific Reports* 16:29905, 2026. doi:10.1038/s41598-026-48763-1.

[25] L. Ding, J. Thomson, J. Taylor, and C. Do. LLMs can assist with proposal selection at large user facilities. *Scientific Reports* 16:29879, 2026. doi:10.1038/s41598-026-60226-1.

[26] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962, 2013.

[27] W. W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic* 32(2), 1967.

[28] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[29] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[30] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. *SSS*, 2011.

[31] R. J. van Glabbeek. The linear time–branching time spectrum I. In *Handbook of Process Algebra*, Elsevier, 2001.

[32] P. Wadler. Propositions as sessions. *ICFP*, 2012.

[33] G. Winskel. Event structures. In *Advances in Petri Nets*, LNCS 255, Springer, 1987.

[34] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science* 34(1–2), 1984.

[35] Flyxion. Spherepop specification. Unpublished manuscript, 2025–2026.

[36] Flyxion. The algebra of binding. Unpublished manuscript, 2026.

[37] Flyxion. The witness plane: admissibility, verification, and commitment in event systems. Unpublished manuscript, 2026.