Session Type State Spaces Form Lattices

Alexandre Zua Caldeira
Independent Researcher, Berlin, Germany
zua@bica-tools.org

We prove that the state space of every well-formed session type, quotiented by strongly connected components, forms a bounded lattice; n-ary parallel composition yields product lattices. Two consequences follow: duality preserves the lattice up to isomorphism, and Gay–Hole width subtyping corresponds to lattice embedding for non-recursive types. We validate this on 108 benchmark protocols across networking, databases, distributed systems, AI, and fault tolerance: all form lattices, 93 distributive, 15 non-distributive. Mechanised in Lean 4 with two independently developed tool implementations.

1 Introduction

Session types [15, 32] describe the communication protocols that govern interaction with objects, channels, or services. A session type specifies the allowed sequences of operations — method calls, branches, selections — together with their ordering constraints. The *state space* of a session type is a labelled transition system (LTS) *over states* obtained by "executing" the type: each state represents a protocol stage, and each transition a permitted action. The semantics is specification-level rather than operational: states are the unit of reachability, not reduction on terms.

State spaces are routinely used for verification, subtype checking, and protocol analysis, yet their algebraic structure has received little attention. Gay and Hole [13], for instance, study subtyping as a preorder on session types but do not investigate the structure of the underlying state spaces. In this paper we show that *the state space of every well-formed session type, after quotienting by strongly connected components, is a bounded lattice*. We call this lattice a *reticulate* — from Latin *reticulatus*, the root of "lattice", giving session type lattices their own name while remaining etymologically precise.

The lattice structure arises naturally from the session type constructors: branch (&) and selection ($\oplus$) create shared extrema that serve as meets and joins; recursion introduces cycles absorbed by the *strongly-connected-component (SCC) quotient* — the standard graph-theoretic operation that collapses each set of mutually reachable states into one equivalence class, yielding a directed acyclic graph (DAG); and n-ary parallel composition ($\|$) yields n-dimensional product lattices. The $\|$ constructor is where lattice structure becomes *necessary* rather than merely convenient. Without $\|$, the SCC quotient of a session type's state space is a tree (pure sequencing), a diamond (binary choice), or a DAG (nested choice). With $\|$, the n-fold product of state spaces inherently has meets and joins corresponding to fork and synchronisation points of concurrent execution.

Our contributions are: (1) the *Reticulate Theorem*: the SCC quotient of every well-formed session type's state space is a bounded lattice, by structural induction (Section 3); (2) *duality preservation*: swapping branch and selection preserves state-space isomorphism (Section 4.1); (3) *subtyping as lattice embedding*: Gay–Hole width subtyping is a label-preserving simulation on the state space whose SCC-quotient projection is a sound lattice embedding for non-recursive types, with the selection case derived from the branch case via duality (Section 4.2); (4) *validation* (Section 6): an empirical study on 108 benchmark protocols using two independently developed tools; mechanised verification of the proof

To appear in EPTCS.

in Lean 4 (17 modules, zero `sorry`); and a distributivity classification: N_5 from depth-unequal choice, M_3 from wide choice ($n \geq 3$), which together account for all 15 non-distributive cases.

Section 2 recalls the relevant background. Section 3 defines the state-space construction and presents the proof. Section 4 develops the duality and subtyping applications. Section 5 demonstrates each result on a concrete concurrent file protocol. Section 6 reports the implementation, the Lean 4 mechanisation, and the empirical study on 108 protocols. Section 7 discusses related work and Section 8 concludes.

2 Preliminaries

2.1 Session Types

The grammar below targets method-call protocols on a single shared *object*; channel-based bi- and multiparty communication (send/receive, delegation, as in process calculi [15]) is out of scope unless the channel is itself modelled as an object, in which case the present framework applies directly. We restrict attention to finite-state (regular) session types.

The grammar below extends typestate session types [14, 19] with parallel composition, letting the lattice construction reach objects shared by concurrent clients:

$$
\begin{array}{rcll}
S & ::= & \&\{m_1 : S_1,\ \ldots,\ m_n : S_n\} & \text{(branch, } n \geq 0\text{)} \\
& \mid & \oplus\{l_1 : S_1,\ \ldots,\ l_n : S_n\} & \text{(selection, } n \geq 0\text{)} \\
& \mid & S_1 \parallel \cdots \parallel S_n & \text{(parallel, } n \geq 2\text{)} \\
& \mid & \mu X.\, S & \text{(recursion)} \\
& \mid & X & \text{(variable)} \\
& \mid & \mathbf{end} & \text{(termination)} \\
D & ::= & S \mid S,\ X_1 = S_1,\ \ldots,\ X_n = S_n & \text{(declaration, equations } n \geq 0\text{)}
\end{array}
$$

A declaration D is a root session type S optionally followed by named equations $X_i = S_i$. The equations decompose a protocol into named, reusable phases; cyclic references express recursion (including mutual recursion), subsuming $\mu X.\, S$ as the special case $X = S$. We retain $\mu X.\, S$ as a separate primitive, equivalent to the single-equation declaration $X = S$.

The branch $\&\{m_1 : S_1, \ldots, m_n : S_n\}$ represents a state in which the object unconditionally offers n method calls; the client exercises an external choice of which to invoke, and the protocol continues as S_i for the chosen m_i ($n = 0$ is the empty offer). In contrast, the selection $\oplus\{l_1 : S_1, \ldots, l_n : S_n\}$ represents an internal choice in which the object returns one of n enumeration labels l_i; the client must dispatch on the returned label and proceed with the corresponding continuation S_i ($n = 0$ is the empty internal choice). Parallel composition $S_1 \parallel \cdots \parallel S_n$ is the flat n-ary form (no nesting), with each S_i describing one client's interactions with the shared object; the sub-protocols run concurrently and may cooperate through that object, with clients playing dual or complementary roles (e.g., reader/writer, publish/subscribe). A variable X is a reference to its binder, either a $\mu X.\, \cdot$ form or an equation $X = S$.

The constant **end** is the explicit termination primitive, not synonymous with the empty branch $\&\{\}$: $\&\{\}$ merely denotes a state with no enabled methods. Every terminated session has no methods, but the converse fails — a parallel branch that has exhausted its actions has no methods to offer yet is not "terminated" until its siblings complete. We retain **end** as a primitive to preserve this asymmetry, matching the additive units of classical linear logic [33]: $\&\{\} \cong \top$, $\oplus\{\} \cong \mathbf{0}$, dual via $\mathbf{0}^{\perp} = \top$.

Definition 1 (Well-formedness). A session type (or declaration) is *well-formed* if all of the following hold.

1. **Closedness.** S (resp. D) has no free variables, where binders are $\mu X.\cdot$ and equation left-hand sides X.
2. **Contractiveness** (also called *guardedness* in the process-calculus tradition). Every recursion $\mu X.S$ has at least one constructor between $\mu X.\cdot$ and each occurrence of X in S; in equation systems, every cyclic reference path passes through at least one constructor.
3. **Termination.** From every state, there is a path to **end**.
4. **Parallel closedness.** In every parallel composition $S_1 \parallel \cdots \parallel S_n$ occurring in S, no variable bound outside the parallel occurs free in any S_i.

Closedness, contractiveness, and parallel closedness are standard in the session-type literature [15, 32, 19, 14] and rule out genuinely ill-formed types — dangling references, uncontractive recursions like $\mu X.X$, and parallel branches that capture outer recursion variables (Lemma 14). Termination is specific to our setting: it is necessary for the bounded-lattice property (Theorem 15), not for admissibility. Non-terminating types remain typeable, with both tools exposing identical `--non-termination` handling (Section 6).

2.2 Lattice Theory

We recall the standard lattice-theoretic vocabulary used in the proofs of Theorem 15; see [4] for a textbook treatment.

Bounded lattice. A bounded lattice is a poset $(L, \leq)$ with a least element $\bot$ and a greatest element $\top$ such that every pair of elements has a *meet* (greatest lower bound) $a \wedge b$ and a *join* (least upper bound) $a \vee b$.

Product lattice. Given bounded lattices $(L_1, \leq_1), \ldots, (L_n, \leq_n)$, their product $L_1 \times \cdots \times L_n$ is ordered componentwise: $(a_1, \ldots, a_n) \leq (b_1, \ldots, b_n)$ iff $a_i \leq_i b_i$ for all i. Meets, joins, top, and bottom are componentwise, and the product of finitely many bounded lattices is itself a bounded lattice.

Distributive lattice. A bounded lattice is *distributive* if $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ holds for all elements; equivalently, if it contains no sublattice isomorphic to N_5 (the pentagon) or M_3 (the diamond with three atoms).

Upward-closed subset. A subset D of a poset $(L, \leq)$ is *upward-closed* if $x \in D$ and $x \leq y$ imply $y \in D$; it is *proper* when $D \neq L$. The dual notion is *downward-closed*.

SCC quotient. A *strongly connected component* (SCC) of a directed graph is a maximal set of mutually reachable vertices; the *SCC quotient* collapses each SCC to a single node, yielding a DAG [29]. In our setting, SCCs collect the states a recursive protocol can revisit; on the quotient, reachability defines a partial order, lifted to a bounded lattice in Lemma 12.

3 State Space Construction and the Reticulate Theorem

The construction is a two-step pipeline:

$$\underset{\text{type}}{S} \xrightarrow{\text{construct}} \underset{\substack{\text{state space}\\ \text{(LTS, cyclic)}}}{\mathscr{L}(S)} \xrightarrow{\substack{\text{order by reachability,}\\ \text{then SCC quotient}}} \underset{\substack{\text{bounded lattice}\\ \text{the } \textit{reticulate}}}{\mathscr{L}(S)/\equiv}$$

The two steps are the first two subsections: the state space is built by structural induction (Section 3.1); then, after its states are ordered by reachability, the SCC quotient collapses it into the bounded lattice (Section 3.2). Section 3.3 then composes the per-constructor lemmas into the headline theorem (Theorem 15). Proof sketches appear inline; full proofs are in Appendix A.

3.1 State spaces and the inductive construction

Definition 2 (State space). A state space is a tuple $(Q, \Sigma, \delta, q_\top, q_\bot)$ where Q is a finite set of states, Σ is a finite alphabet of action labels, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q_\top \in Q$ is the initial state, and $q_\bot \in Q$ is the terminal state. The construction of Definition 3 assigns a state space $\mathscr{L}(S)$ to each well-formed session type S.

Definition 3 (State-space construction $\mathscr{L}(S)$). The state space $\mathscr{L}(S) = (Q, \Sigma, \delta, q_\top, q_\bot)$ is defined by structural induction on closed well-formed S (Definition 1). The induction yields state spaces, not bounded lattices. Of the grammar's two base cases, **end** and X, only **end** takes a clause below: under closedness every bare X is bound (by an enclosing $\mu X.\cdot$ or a definition $X = S'$), so each occurrence is built as a *placeholder* state and then eliminated by cases (4)–(5), which redirect it as a back-edge to the already-built entry state. For closed well-formed S, the SCC quotient $\mathscr{L}(S)/\equiv$ is a bounded lattice (Definition 6 and Theorem 15).

1. **End.** $\mathscr{L}(\mathbf{end}) = (\{q_0\}, \emptyset, \emptyset, q_0, q_0)$ — a single state, both initial and terminal.
2. **Branch** $\&\{m_i : S_i\}_{i=1..n}$.

$$\begin{aligned} Q &= \{q_0\} \cup \textstyle\bigcup_i (Q_i \setminus \{q_\bot^i\}) \cup \{q_\bot\}, \\ \Sigma &= \{m_1, \ldots, m_n\} \cup \textstyle\bigcup_i \Sigma_i, \\ \delta &= \{(q_0, m_i, q_\top^i)\}_i \cup \textstyle\bigcup_i \delta_i[q_\bot^i := q_\bot], \end{aligned}$$

   where $Q_i, \Sigma_i, \delta_i, q_\top^i, q_\bot^i$ are the components of $\mathscr{L}(S_i)$; the substitution $[q_\bot^i := q_\bot]$ shares $q_\bot$ across children; $q_\top = q_0$. For $n = 0$, $Q = \{q_0, q_\bot\}$, $\Sigma = \{\tau\}$, $\delta = \{(q_0, \tau, q_\bot)\}$ with τ a silent action for vacuous termination; the SCC quotient is the chain $q_0 > q_\bot$.
3. **Selection** $\oplus\{l_i : S_i\}_{i=1..n}$. Identical tuple to branch at the state-space level: the construction depends only on the continuations (Lemma 9), not on polarity.
4. **Recursion** $\mu X.S'$. Build $\mathscr{L}(S') = (Q', \Sigma', \delta', q'_\top, q'_\bot)$ with fresh placeholder states for each occurrence of X; let $P_X \subseteq Q'$ collect these placeholders. Then $Q = Q' \setminus P_X$, $\Sigma = \Sigma'$, $\delta = \delta'[P_X := q'_\top]$ (redirect every transition targeting a placeholder to $q'_\top$, creating back-edges), $q_\top = q'_\top$, $q_\bot = q'_\bot$.
5. **Named definitions** $D = S, X_1 = S_1, \ldots, X_k = S_k$ (root session type followed by $k \geq 0$ equations). Build $\mathscr{L}(S)$ and each $\mathscr{L}(S_j)$ with placeholders $p_{i,j}$ for each occurrence of X_j. For each name X_j, replace every transition targeting any $p_{*,j}$ with a transition to $q_\top^{S_j}$, and remove the placeholders. The result is $\mathscr{L}(D) = (Q, \Sigma, \delta, q_\top^S, q_\bot^S)$ where $Q = (Q^S \cup \bigcup_j Q^{S_j})$ with placeholders removed, $\Sigma = \Sigma^S \cup \bigcup_j \Sigma^{S_j}$, and δ is the global redirection.

6. **Parallel** $S_1 \parallel \cdots \parallel S_n$. $\mathscr{L}(S_1 \parallel \cdots \parallel S_n) = (Q_1 \times \cdots \times Q_n, \Sigma, \delta_\times, (q_\top^1, \ldots, q_\top^n), (q_\bot^1, \ldots, q_\bot^n))$ with $\Sigma = \bigcup_i \Sigma_i$ and $\delta_\times$ the componentwise advance relation: $(s_1, \ldots, s_i, \ldots, s_n) \xrightarrow{a} (s_1, \ldots, s_i', \ldots, s_n)$ for each $s_i \xrightarrow{a} s_i'$ in δ_i.

Inductive, not coinductive. Definition 3 is structural recursion on the finite six-constructor syntax; the carrier is state spaces, not bounded lattices. Recursion uses a back-edge rather than unfolding: for $\mu X.S$ the body S is built once and each occurrence of X is redirected to the top $q_\top$ (cases 4–5), forming a finite cycle. The construction never forms the infinite unfolding — the greatest fixed point — and uses no coinduction; the SCC quotient collapses the cycle. The Lean mechanisation is likewise coinduction-free.

3.2 SCC quotienting and the bounded lattice

A bounded lattice is a poset with a least and a greatest element and a meet and a join for every pair (Section 2.2). We establish these for $\mathscr{L}(S)/\equiv$: reachability orders it as a poset (Definitions 4 and 6); Proposition 7 gives its least and greatest elements; and the meets and joins are supplied constructor by constructor, by structural induction on S, assembled by the Reticulate Theorem (Theorem 15).

Definition 4 (Reachability). Write $s_1 \twoheadrightarrow s_2$ ("s_1 reaches s_2") iff there exists a path from s_1 to s_2 in (Q, δ).

Proposition 5 (Preorder). *The relation $\twoheadrightarrow$ is reflexive and transitive, hence a preorder; cycles from recursion prevent antisymmetry.*

Definition 6 (SCC quotient and its order). The *SCC quotient* $\mathscr{L}(S)/\equiv$ identifies mutually reachable states, collapsing each recursion cycle to one class and yielding an antisymmetric quotient. It is ordered by $[s_1] \leq [s_2]$ iff $s_2 \twoheadrightarrow s_1$: a state is smaller when less of the protocol remains, so execution descends from $q_\top$ to $q_\bot$.

Proposition 7 (Extrema). *For every well-formed S, the initial state $q_\top$ reaches every state and the terminal state $q_\bot$ is reachable from every state; hence the quotient $\mathscr{L}(S)/\equiv$ has greatest element $q_\top$ and least element $q_\bot$.*

It remains to exhibit a meet and a join for every pair. Each grammar constructor corresponds to a lattice operation on its sub-lattices: **end** yields the one-element lattice; choice glues sub-lattices through a fresh top into a diamond; recursion absorbs an upward-closed set into its body's top via the SCC quotient; named definitions iterate that absorption; parallel takes a Cartesian product. Three structural innovations make this work jointly: *end-identification* (singleton $\mathscr{L}(\mathbf{end})$ with shared $q_\bot$ across branch and selection children; Lemma 8, Lemma 10), the *SCC quotient* (Definition 6; Lemma 12, Lemma 13), and the $\parallel$-*product* (Lemma 14). Each is load-bearing on at least one constructor case. Each of the following lemmas treats one constructor, assuming the sub-state-spaces are bounded lattices and extending the property to the composite; Theorem 15 composes the cases.

Lemma 8 (End). $\mathscr{L}(\mathbf{end})/\equiv$ *is a bounded lattice.*

Lemma 9 (Polarity erasure). *For any labels $a_1, \ldots, a_n$ and well-formed types $S_1, \ldots, S_n$,*

$$\mathscr{L}(\&\{a_1{:}S_1, \ldots, a_n{:}S_n\}) \;=\; \mathscr{L}(\oplus\{a_1{:}S_1, \ldots, a_n{:}S_n\}).$$

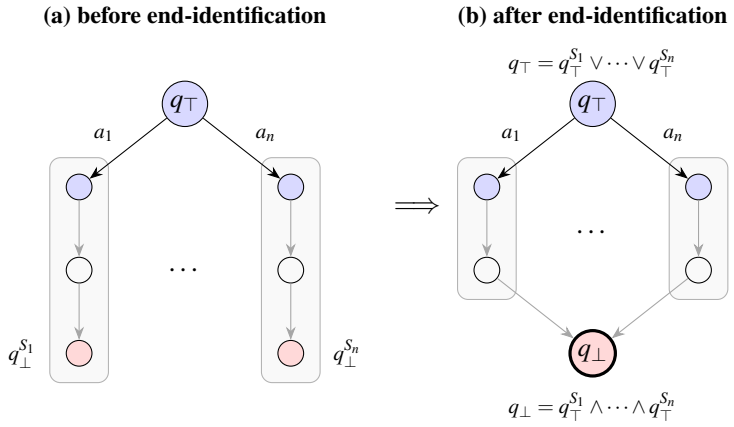

Figure 1: Constructing $\mathscr{L}(\&\{a_1{:}S_1,\ldots,a_n{:}S_n\})$. (a) A fresh $q_\top$ fans by one transition a_i into each sub-lattice $\mathscr{L}(S_i)$, each with its own least element $q_\bot^{S_i}$. (b) End-identification glues every $q_\bot^{S_i}$ onto the single reserved terminal $q_\bot$; the fresh $q_\top$ is then the only common upper bound and $q_\bot$ the only common lower bound, making the disjoint sub-lattices one bounded lattice. For $n = 2$ with singleton arms this is the diamond lattice M_2. The join and meet annotations in (b) assume $n \geq 2$; at $n = 1$ the fresh $q_\top$ simply prepends one element above $\mathscr{L}(S_1)$.

Proof. Both constructions create a fresh $q_\top$ with n transitions to sub-state-space entries, sharing $q_\bot$ — depending only on the continuations, not on the polarity. Polarity is a typing discipline, not a structural property of the state space. □

Lemma 10 (Choice). *If $\mathscr{L}(S_i)/{\equiv}$ is a bounded lattice for each i, then so is $\mathscr{L}(\&\{a_1 : S_1,\ldots,a_n : S_n\})/{\equiv}$; by Lemma 9 the same holds for $\oplus\{a_1 : S_1,\ldots,a_n : S_n\}$.*

Proof. Adjoining a common top $q_\top$ and bottom $q_\bot$ turns the sub-lattices into a single bounded lattice: meets and joins within a sub-lattice are inherited, and across sub-lattices, which are disjoint above $q_\bot$, $q_\top$ is the only common upper bound and $q_\bot$ the only common lower bound (Figure 1). □

Three lemmas cover recursion and named equations. Lemma 11 is the order-theoretic tool: collapsing a proper upward-closed set onto $q_\top$ preserves boundedness. Lemma 12 applies it to a single recursive binder, whose back-edges form an SCC absorbed into $[q_\top]$ (Figure 2). Lemma 13 treats named declarations, which are more expressive than $\mu X.S$: cyclic references (single or mutual recursion) reuse the same absorption, while an acyclic shared name — like **end** — collapses duplicated subtrees, giving lattice rather than tree structure.

Lemma 11 (Top absorption). *Let $(L,\leq)$ be a bounded lattice and D a* proper *upward-closed subset of L containing $\top$ (equivalently, $\bot \notin D$). Define $L' = (L \setminus D) \cup \{\top\}$ with the restricted ordering. Then L' is a bounded lattice.*

Proof. Since D is upward-closed and proper, $\bot \notin D$, so L' retains both extrema. Meets in $L \setminus D$ stay in $L \setminus D$ (the complement of an upward-closed set is downward-closed) and are preserved. Joins in $L \setminus D$ are likewise preserved, except those whose L-join lies in D, which collapse to $\top$ in L' — any non-$\top$ upper bound would itself lie in D by upward-closure. □

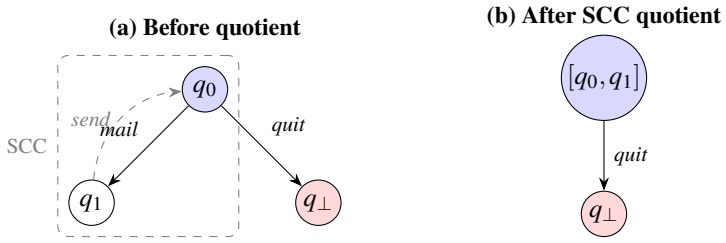

Figure 2: SMTP session type $\mu X.\&\{mail : \&\{send : X\},\ quit : \mathbf{end}\}$. (a) Before quotient: the recursion redirects *send* from the placeholder to q_0, creating a cycle $q_0 \to q_1 \to q_0$ (dashed). (b) After SCC quotient: the cycle collapses into $[q_0, q_1]$, yielding a 2-element chain. Here $D = \{q_0, q_1\}$ is upward-closed and omits $q_\perp$, so Lemma 11 applies; the tool confirms 3 states, 3 transitions.

Lemma 12 (Recursion). *Let S be a body with a single free variable X, such that the state space $\mathscr{L}(S)$ with placeholder state p_X satisfies: (i) $q_\top^S$ reaches every non-placeholder state; (ii) the placeholder-free SCC quotient is a bounded lattice; and (iii) the recursion $\mu X.S$ satisfies the **Termination** clause of Definition 1. Then $\mathscr{L}(\mu X.S)/{\equiv}$ is a bounded lattice.*

Proof. Adding back-edges from each placeholder p_X to $q_\top^S$ collapses these states with $q_\top^S$ into one SCC class. The set D of quotient classes absorbed into $[q_\top^S]$ is upward-closed (mutual reachability through $q_\top^S$ propagates upward), and the Termination clause (Definition 1) ensures $[q_\perp] \notin D$, making D a *proper* upward-closed subset of the body quotient. Lemma 11 then yields the lattice. □

Lemma 13 (Named definitions). *Let $D = S, X_1 = S_1, \ldots, X_k = S_k$ be a well-formed declaration in which every equation is used either for* recursion *(each X_j lies on a cyclic reference path) or* at most once*. If $\mathscr{L}(S)/{\equiv}$ and each $\mathscr{L}(S_i)/{\equiv}$ are bounded lattices, then $\mathscr{L}(D)/{\equiv}$ is a bounded lattice. More generally it suffices that the sharing be* non-reconvergent*: no non-recursive name is reached from two distinct branches of the same choice.*

Proof. Three cases, by how each name is used. A *cyclic* reference behaves like recursion: its back-edges merge states into an SCC, which Lemma 11 collapses (one absorption per SCC, in any order), exactly as in Lemma 12. A *non-recursive name used along a single path* inlines its sub-lattice beneath the one state that references it, adding no new incomparable pairs, so every meet and join survives. The remaining case is *reconvergence*: when one name is reached from two distinct branches of a choice, those branches rejoin at the name's entry, leaving two incomparable states with two incomparable common lower bounds, so their meet does not exist. Such declarations are only bounded *partial* lattices, handled by the Reticulate Theorem (Section 3.3). □

Lemma 14 (Parallel). *If $\mathscr{L}(S_i)/{\equiv}$ is a bounded lattice for each $i \in \{1, \ldots, n\}$, then $\mathscr{L}(S_1 \parallel \cdots \parallel S_n)/{\equiv}$ is a bounded lattice.*

Proof. By the parallel case of Definition 3 and the independence of the n branches (Termination ensures every branch reaches **end**; Parallel closedness ensures no shared variables), the product inherits SCC structure componentwise: $\mathscr{L}(S_1 \parallel \cdots \parallel S_n)/{\equiv} \cong \prod_i \mathscr{L}(S_i)/{\equiv_i}$. The product of finitely many bounded lattices is itself a bounded lattice (Section 2.2), illustrated for $n = 2$ in Figure 3. □

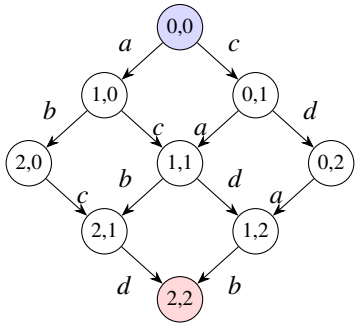

Figure 3: Product lattice $\mathscr{L}(\&\{a : \&\{b : \mathbf{end}\}\} \parallel \&\{c : \&\{d : \mathbf{end}\}\})$. The 9 states form a 3×3 grid lattice with componentwise ordering.

3.3 The Reticulate Theorem

The per-constructor lemmas now compose into the headline result, followed by tightness and a remark on realisability.

Theorem 15 (Reticulate Theorem). *For every well-formed session type S over the six constructors of Section 2.1 (with single or mutual recursion), $\mathscr{L}(S)/\equiv$ is a bounded lattice.*

Proof. By structural induction on S, applying the per-constructor Lemmas 8, 10 and 12 to 14. Each case preserves the bounded-lattice property, completing the induction. □

The theorem is stated for session types; for declarations with *named equations* the construction yields a bounded *partial* lattice in general. When two equations are each referenced from two distinct choice branches, the branches reconverge and a meet or join may fail to exist; recursive sharing is absorbed by the SCC quotient and stays a lattice. Characterising the sharing patterns that keep the operations total is open (Section 8).

Corollary 16 (Termination tightness). *The **Termination** clause is tight: relaxing the exit-path condition yields types whose SCC-quotiented state spaces are not lattices.*

A witness: $\&\{a : \mathbf{end},\ b : \mu X.\,\&\{a : X\}\}$ violates **Termination** — from the body root $\&\{a : X\}$, the only outgoing transition unfolds X back to the same root, so no path to **end** exists. After SCC quotient the recursive branch collapses to a single class with no outgoing transition to a terminal, leaving no bottom element reachable from all states.

Proposition 17 (A realisable fragment). *Every finite chain is a reticulate, and reticulates are closed under finite products, $\mathscr{L}(S_1 \parallel \cdots \parallel S_n)/\equiv \;\cong\; \prod_i \mathscr{L}(S_i)/\equiv$. Hence every finite product of chains is a reticulate.*

Proof. A single-arm branch prepends a fresh top above its continuation (Lemma 10 at $n = 1$), so k nested single-arm branches over $\mathscr{L}(\mathbf{end})$ realise the $(k+1)$-element chain; closure under finite products is Lemma 14. □

4 Duality and Subtyping as Lattice Consequences

The lattice structure has two consequences for class session types: duality is invisible at the lattice level (Section 4.1); width subtyping reduces to lattice embedding (Section 4.2). On *channel* session types, duality is operational, swapping $\&/\oplus$ together with $?/!$ [14]; for *class* session types, the residual polarity swap preserves lattice structure (Proposition 18).

4.1 Duality Preserves Lattice Isomorphism

The *dual* of a session type swaps branch and selection: $\mathit{dual}(\&\{m_i : S_i\}) = \oplus\{m_i : \mathit{dual}(S_i)\}$ and vice versa; all other constructors (**end**, $\|$, $\mu X.\cdot$) are covariant. Duality is an involution: $\mathit{dual}(\mathit{dual}(S)) = S$.

Proposition 18 (Duality isomorphism). *For every well-formed session type S, there is a bounded-lattice isomorphism*

$$\mathscr{L}(S)/{\equiv} \;\cong\; \mathscr{L}(\mathit{dual}(S))/{\equiv}.$$

Proof. By structural induction on S. The base case (**end**) is trivial since duality is the identity. For choice, branch and selection produce identical state spaces (Lemma 9), so polarity reversal is invisible at the lattice level. For parallel, recursion, and named definitions, duality distributes through the constructors, and the inductive isomorphisms on sub-types compose into one on the whole. □

Proposition 18 says that polarity is invisible at the lattice level: S and $\mathit{dual}(S)$ produce the same lattice structure. For class session types, only the object's protocol S is explicit; the dual $\mathit{dual}(S)$ specifies the implicit second-party (client) protocol, and these two faces share the same lattice structure. Both tools verify $\mathscr{L}(S)/{\equiv} \cong \mathscr{L}(\mathit{dual}(S))/{\equiv}$ for all 108 benchmarks. The Lean 4 mechanisation proves $\mathit{dual}(\mathit{dual}(S)) = S$ on the AST; the state-space isomorphism follows from the constructor-wise argument above.

The construction erases polarity (Lemma 9), so the reticulate does not distinguish S from $\mathit{dual}(S)$; Proposition 18 thus reflects this erasure rather than a structural symmetry. Recovering polarity — by labelling the cover relation — would make duality a non-identity involution and realise the linear-logic reading (& as meet $\sqcap$, $\oplus$ as join $\sqcup$); this is left to future work on labelled reticulates.

4.2 Subtyping as Lattice Embedding

The lattice structure has a concrete consequence for subtype checking. Gay and Hole [13] define a coinductive subtyping relation $\leq_{GH}$ on session types as the greatest fixed point of a monotone operator on type-indexed binary relations. In words, $S_1 \leq_{GH} S_2$ means the subtype S_1 can be substituted wherever a S_2-typed object is expected — a branch subtype offers *more* methods (covariant width); a selection subtype emits *fewer* labels (contravariant width); parallel is congruent; recursion is closed under one-step unfolding. We refer to [13] for the formal inference rules.

The contravariance of selection width is the *dual* of branch covariance: by Proposition 18, swapping polarity preserves the lattice, so the selection case follows from the branch case applied to the dual types. Since branch and selection produce *identical state spaces* (Section 3), the embedding relationship depends on which type has more top-level choices — not on the polarity.

Definition 19 (Lattice embedding). An *embedding* of a bounded lattice L_1 into L_2 is an injective map $\varphi : L_1 \to L_2$ that preserves and reflects order: $a \leq b$ iff $\varphi(a) \leq \varphi(b)$. Every embedding preserves meets and joins.

On the bare order this embedding is *sound* for $\leq_{GH}$ but not faithful: types differing only in terminal-method width — e.g. $\&\{a : \mathbf{end}\}$ and $\&\{a : \mathbf{end},\ b : \mathbf{end}\}$ — share one lattice, because *end*-identification sends both methods to the common $q_\perp$. A faithful invariant needs the labelled transitions δ: subtyping is a label-preserving simulation, of which the order embedding is the SCC-quotient projection. The labelled-lattice / bisimulation correspondence is developed separately.

Proposition 20 (Width–embedding correspondence). *For all well-formed non-recursive session types related by* width *subtyping (shared-label continuations coincide, $S_i = S'_i$ for $i \leq k$):*

1. ***Branch width.*** *If $\&\{m_1{:}S_1, \ldots, m_n{:}S_n\} \leq_{GH} \&\{m_1{:}S'_1, \ldots, m_k{:}S'_k\}$ with $k \leq n$, then the supertype embeds into the subtype:*

$$\mathscr{L}(\&\{m_1{:}S'_1, \ldots, m_k{:}S'_k\})/{\equiv} \hookrightarrow \mathscr{L}(\&\{m_1{:}S_1, \ldots, m_n{:}S_n\})/{\equiv}.$$

2. ***Selection width.*** *If $\oplus\{l_1{:}S_1, \ldots, l_n{:}S_n\} \leq_{GH} \oplus\{l_1{:}S'_1, \ldots, l_k{:}S'_k\}$ with $n \leq k$, then the subtype embeds into the supertype:*

$$\mathscr{L}(\oplus\{l_1{:}S_1, \ldots, l_n{:}S_n\})/{\equiv} \hookrightarrow \mathscr{L}(\oplus\{l_1{:}S'_1, \ldots, l_k{:}S'_k\})/{\equiv}.$$

Proof. Without recursion the state space is acyclic, so the SCC quotient is the identity and no states merge. For width subtyping the shared continuations coincide, so the supertype's reticulate is exactly the subtype's restricted to the shared labels; the extra labels contribute fresh sub-lattices disjoint above $q_\perp$ (Lemma 10). This inclusion is injective, order-preserving and order-reflecting. Selection-width follows by duality (Proposition 18). □

The two clauses share one principle: the side with fewer top-level choices has the smaller lattice and embeds into the larger. The flip in direction — supertype-into-subtype for &, the reverse for $\oplus$ — is not a property of the lattice (which by Lemma 9 ignores polarity) but of $\leq_{GH}$, under which a branch subtype offers more labels and a selection subtype fewer.

Proposition 20 isolates *width* subtyping; the full relation $\leq_{GH}$ also allows *depth* subtyping ($S_i \leq_{GH} S'_i$ on shared labels). Each such continuation contributes its own embedding, so $\leq_{GH}$ corresponds to a *family* of embeddings rather than a single map; a complete lattice-theoretic characterisation, including recursion, is ongoing work. Both tools implement the embedding check, and Section 5 demonstrates it on a concrete protocol.

5 Case Study: A Concurrent File Protocol

We demonstrate Theorem 15, Proposition 18, and Proposition 20 on a sequential file reader, then recover its writer counterpart and compose the two in parallel into the canonical concurrent read/write File.

Construction. The FileReader protocol — a single thread reading from open to close, with each read returning *data* until *eof* — is

$$S_{FileReader} = \&\{open : \mu X.\,\&\{read : \oplus\{data : X,\ eof : Close\}\}\}, \quad Close = \&\{close : \mathbf{end}\}.$$

The bottom-up construction matches subterms to constructor lemmas:

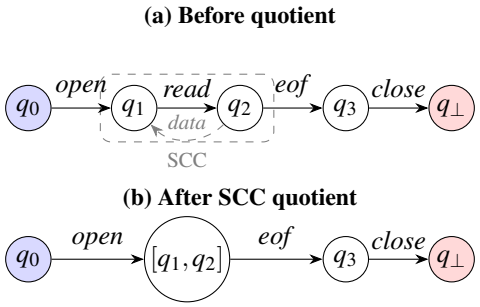

Figure 4: $\mathscr{L}(S_{FileReader})$: (a) before quotient, with the recursion edge $q_2 \to q_1$ dashed; (b) after the SCC quotient $[q_1, q_2]$, a 4-element chain.

Subterm	Lemma
end	Lemma 8
$Close = \&\{close : \mathbf{end}\}$	Lemma 10, n=1; Lemma 13
$\oplus\{data : X,\ eof : Close\}$	Lemma 10, n=2
$\&\{read : \ldots\}$	Lemma 10, n=1
$\mu X. \ldots$	Lemma 12, Lemma 11
$\&\{open : \ldots\}$	Lemma 10, n=1

The recursion step makes q_1, q_2 mutually reachable, so Lemma 11 collapses them into the class $[q_1, q_2]$: the SCC captures the *read/data loop*, the protocol cycling between read and selection indefinitely before choosing *eof*. By Theorem 15, the composition witnesses that $\mathscr{L}(S_{FileReader})/\equiv$ is the bounded lattice $q_0 > [q_1, q_2] > q_3 > q_\perp$ shown in Figure 4(b).

Duality. The dual swaps every choice polarity:

$$dual(S_{FileReader}) = \oplus\{open : \mu X.\ \oplus\{read : \&\{data : X,\ eof : \oplus\{close : \mathbf{end}\}\}\}\}.$$

Each choice flips agency: methods that $S_{FileReader}$ *offers* are *invoked* by $dual(S_{FileReader})$, and outcomes the file *internally selects* are *received* by the dual — the same protocol read from the opposite endpoint. By Lemma 9, replaying Definition 3 yields the same Hasse diagram (Figure 4(b)), witnessing Proposition 18.

Subtyping. Consider a richer file protocol that also lets a client read metadata and close without entering the read loop:

$$S^{+}_{FileReader} = \&\{open : \mu X.\&\{read : \oplus\{data : X,\ eof : \&\{close : \mathbf{end}\}\}\},\ stat : \&\{close : \mathbf{end}\}\}$$

By branch covariance (Gay–Hole width subtyping), $S^{+}_{FileReader} \leq_{GH} S_{FileReader}$: the subtype offers *more* top-level methods. Applying Definition 3, the outer branch widens the top q_0 with a second transition $q_0 \xrightarrow{stat} q_4$, where $q_4 = \&\{close : \mathbf{end}\}$ is a fresh state reached *outside* the read loop; the shared *open* arm reproduces Steps 1–6 unchanged. After SCC quotient the subtype's reticulate is the five-element lattice

$$q_0 > [q_1, q_2] > q_3 > q_\perp, \qquad q_0 > q_4 > q_\perp,$$

with top q_0 and the two arms meeting at $q_\perp$ (q_3, q_4 incomparable). The supertype's 4-chain embeds into it as the *open* arm, q_4 lying outside the image — a strict, order-preserving and -reflecting embedding. This type is recursive, so it lies outside Proposition 20; here the recursion sits in the shared *open* continuation, so the embedding survives.

FileWriter. The symmetric writer arises by renaming *read*/*data* to *write*/*ack*:

$$S_{FileWriter} = \&\{open : \mu X.\,\&\{write : \oplus\{ack : X,\ eof : \&\{close : \mathbf{end}\}\}\}\}.$$

The construction is invariant under method renaming, so $\mathscr{L}(S_{FileWriter}) \cong \mathscr{L}(S_{FileReader})$.

File. The canonical concurrent read/write File bundles both inside one object: $S_{File} \triangleq S_{FileReader} \parallel S_{FileWriter}$. Two threads sharing only a filename each instantiate one arm independently, so two opens and two closes on the same logical file are not errors but transitions on disjoint arms. The reticulate is the componentwise product $\mathscr{L}(S_{FileReader}) \times \mathscr{L}(S_{FileWriter})$ (Lemma 14).

6 Implementation, Mechanisation, and Empirical Validation

Implementation. We have implemented the state-space construction and lattice checking in two independently developed tools targeting distinct deployment surfaces: a Python 3.12 CLI for standalone protocol analysis, and a Java 21 annotation processor that performs the check at compile time on `@Session`-annotated code. Both are thoroughly tested and agree on all 108 benchmarks. The pipeline is identical in both tools: a parser converts the session-type source (Java `@Session` annotations or text input) to an AST; a builder applies Definition 3 clause-by-clause to produce the labelled state space; the SCC quotient is computed via Tarjan's algorithm [29]; and an all-pairs check verifies $\wedge$ and $\vee$ existence on the quotient. The lattice check is $O(n^2 \cdot m)$ where $n = |Q|$ and $m = |\delta|$, completing in under 100 ms on commodity hardware for all benchmarks (a sub-millisecond median, and under 5 ms for the largest benchmark with $|Q| = 41$).

Test generation. A direct application of the lattice goes beyond verification: automated protocol-conformance test generation. Both implementations expose a test-generation mode that emits JUnit 5 cases in three categories: valid paths (chains from $q_\top$ to $q_\bot$, enumerated by bounded DFS), violations (disabled methods at each reachable state), and incomplete prefixes (proper prefixes of valid paths). The lattice's boundedness guarantees a finite test suite; bottom-reachability (Proposition 7) guarantees every valid path terminates. Non-terminating types are accepted under the same three-mode flag `--non-termination={error,warn,off}` as the lattice check; modes, default, and exact wording are pinned identical across both tools by mirrored parity test suites. Valid paths are then empty, but violations and incomplete prefixes remain. Both tools agree on the generated test sets across the 108 benchmarks.

Mechanisation. The paper's results have been mechanically verified in Lean 4 [22] using Mathlib [30]. Theorem 15 is mechanised as a universal theorem `reticulate_lattice` quantifying over every well-formed session type, with no extra hypothesis; duality's AST involution $dual(dual(S)) = S$ and polarity swap are mechanised, and Proposition 20 (width embedding) and the forward direction of Proposition 17 are backed by dedicated modules. The full paper-faithful artefact comprises 17 modules with zero `sorry` and zero `admit`, and uses no coinduction. The mechanisation, the Python checker, the benchmark corpus, and a per-result map to the Lean modules are archived as a companion artifact [34].

Empirical validation. The 108-protocol benchmark suite plays two distinct roles: *(i)* it tests that the Python and Java implementations *conform* to the theorem on real protocols (tooling conformance); and

(ii) it characterises the theorem's empirical scope — every protocol in the suite is well-formed and produces a lattice, and the 86%/14% distributive split (below) gives a first empirical picture of reticulate structure across application domains. The 108 protocols (Table 1) draw from *literature benchmarks* (Java Iterator, SMTP, HTTP, JDBC, OAuth 2.0, Two-Buyer, and others from the Mungo/StMungo/Scribble literature), *industrial protocols* (MCP, A2A, TLS, Saga Orchestrator, WebSocket, gRPC), and *synthetic benchmarks* systematically covering edge cases (deep recursion, wide branching $n \geq 10$, nested parallel, mixed polarity).

Table 1: Benchmark summary by source.

Source	Protocols	Distributive	Non-dist.
Literature	34	32	2
Industrial	26	18	8
Synthetic	48	43	5
Total	**108**	**93**	**15**

All 108 protocols produce bounded lattices under SCC quotient. We classify each lattice in the Birkhoff hierarchy [4]: 93 (86%) are *distributive* (no forbidden N_5 or M_3 sublattice) and 15 (14%) are non-distributive — 13 containing N_5 (pentagon) and 3 containing M_3 (diamond); one protocol (Hex) contains both. All product lattices from parallel composition are distributive, consistent with the lattice-theoretic result that the product of distributive lattices is distributive. This classification is mechanised in Lean 4 with a proof (by `decide`) that the distributive law fails on N_5.

Two structural patterns account for all 15 non-distributive cases: N_5 from *depth-unequal choice* (branches of a choice constructor terminating at different depths yield a 3-chain plus side-branch) and M_3 from *wide choice* ($n \geq 3$ branches reconverging at $q_\perp$ yield three pairwise-incomparable atoms; binary choice produces at most M_2, which is distributive). Both patterns are polarity-symmetric (Proposition 18); the empirical clustering of N_5 in polarity-mixed nestings and M_3 in wide branchings (Table 2) reflects regularities of real protocols, not structural requirements. Characterising precisely which session types yield distributive reticulates remains open.

We additionally enumerate all well-formed terminating session types up to depth 3 with label sets $\{a,b\}$, confirming that every such type produces a lattice — further evidence for universality.

7 Related Work

Session types. Session types were introduced by Honda et al. [15] for binary communication and extended to multiparty settings [16]; Vasconcelos [32], Hüttel et al. [17], and Ancona et al. [2] provide foundational treatments and comprehensive surveys. Honda et al.'s formulation is a process calculus with explicit $?T/!T$ directionality and delegation; ours is an object-protocol specification on a single shared object, with $\|$ over n participants on that object rather than between independent processes. Session types for objects were developed by Dezani-Ciancaglini et al. [12] and Gay et al. [14], with tool support from MUNGO and STMUNGO [19, 20]. MUNGO uses named definitions as the primary design mechanism; our grammar adopts the same notation. By contrast, our $\|$ constructor has no counterpart in typestate tools. Gay and Hole [13] established subtyping for session types (revisited as lattice embedding in Section 4.2). Scalas and Yoshida [27] rebuild multiparty session types without global types or projection, replacing them with a behavioural safety invariant on typing contexts. Across the work above, the focus is on the type algebra; we study the algebraic structure of the *state spaces* induced by individual session types.

Table 2: The 15 non-distributive benchmarks. 13 contain N_5 (pentagon); 3 contain M_3 (diamond).

Protocol	$\lvert Q\rvert$	Type	Cause
TLS Handshake	13	N_5	branch-inside-selection (HELLO_RETRY)
Two-Phase Commit	7	N_5	branch-selection nesting
Ki3 Onboarding	32	N_5	nested branch-selection
Ki3 CI/CD Pipeline	41	N_5	4-stage branch-selection nesting
Quantum Measurement	9	N_5	crossed branch-selection
SSH Handshake	18	N_5	branch-selection nesting
Alternative Splicing	7	M_3	three branches sharing outcomes
QCD Gluon Exchange	6	M_3	three-way symmetric reconvergence
Hex (2×2)	30	both	game-tree reconvergence + nesting

Plus 6 further benchmarks (biology, physics, security, games) with analogous patterns.

Intersection/union session types. Padovani [24, 23] and Acay and Pfenning [1] develop a *type-algebra* lattice where internal and external choice are replaced by intersection $\wedge$ and union $\vee$, yielding a semantic subtyping lattice $\langle 2^{\mathscr{P}}, \subseteq \rangle$ of closed process-set denotations. Our lattice is orthogonal: it lives *inside* a single type, on its reachable state space. Padovani [24] (p. 80) conjectures distributivity of his process-set lattice and leaves it open; our 86%/14% distributive split on 108 benchmarks (Section 6) is a first empirical probe of the analogous question on reachability lattices, with a formal bridge left to future work. Barbanera and de'Liguoro [3] similarly develop a sub-behaviour preorder on client/server session types via LTS compliance, but prove duals-as-minima rather than a lattice. Strom and Yemini's 1986 typestate proposal [28] required typestates of each type to form a meet-semilattice — the closest historical ancestor. But their lattice captures abstract-interpretation dataflow, not protocol reachability; in particular, our $\|$ constructor is what forces completion to a *full* lattice.

Lattices in programming languages. Lattice theory underlies abstract interpretation [7] and is the standard algebraic vocabulary for order in PL [4, 11]. We show that *behavioural* types produce lattices at the state-space level, a distinct setting.

Linear logic and session types. Caires and Pfenning [5] established a Curry–Howard correspondence between session types and intuitionistic linear logic; Wadler's [33] classical version ("propositions as sessions") links negation to session-type duality, complementing our state-space duality (Proposition 18). Toninho, Caires, and Pfenning [31] extended the correspondence to higher-order sessions; the lattice-theoretic perspective on higher-order types remains open.

Subtyping. Gay and Hole [13] defined subtyping for session types; Dardha et al. [10] revisited it in a pi-calculus setting. Padovani [25] extended subtyping to multiparty types with fairness; his coinductive characterisation operates on infinite trees, complementary to our finite quotient embedding. Castagna and Frisch [6] developed semantic subtyping using set-theoretic models. Our characterisation of subtyping as lattice embedding (Section 4.2) provides an algebraic alternative to the coinductive, fair, and set-theoretic approaches.

Multiparty session types. Honda, Yoshida, and Carbone [16] introduced multiparty session types (MPST), where a global type is projected onto local types for each role. Our Reticulate Theorem applies to local types obtained by projection; whether projection itself is a lattice morphism between global and

local reticulates is future work. Dagnino, Giannini, and Dezani-Ciancaglini [8] recently developed deconfined global types in a coinductive, process-calculus setting proving progress and deadlock-freedom; our setting is complementary — single object, finite-state (μ-unfolded), specification-based — proving lattice structure rather than progress.

Process algebra. Our $\|$ operates on a *single shared object* rather than independent processes [21, 26], requiring product lattice construction. Kobayashi [18] and Dardha and Gay [9] use type systems for deadlock-freedom; our lattices complement this by making all interleavings explicit, guaranteeing that every concurrent path reaches $\bot$.

8 Conclusion

We have proved that the state space of every well-formed session type, quotiented by strongly connected components, forms a bounded lattice (Theorem 15). The proof is constructive and modular, with one lemma per constructor; the grammar is parsimonious (six type constructs, three n-ary), and well-formedness reduces to four clauses (Definition 1), three standard in the session-type literature and one (termination) specific to our setting. We validated the theorem on 108 benchmark protocols using two independently developed, thoroughly tested tools, and mechanically verified the core proof in Lean 4 (17 modules, zero `sorry`).

This lattice structure — the *reticulate* — opens several directions. The *distributivity classification* (86% distributive, 14% non-distributive) suggests finer algebraic structure to exploit, such as canonical forms for the distributive fragment. Proposition 17 answers a fragment of the converse, but the reachability order does not always yield a lattice: where sharing reconverges it descends to a bounded partial lattice (Section 3.3). Ordered structure on session types has been observed on the subtyping preorder [13, 3] and on typestate dataflow [28], but a *syntactic characterisation* of which structure a type's reachability order carries — the natural generalisation of the Reticulate Theorem — remains open. More broadly, the construction reads as one direction of an *abstract interpretation*: Theorem 15 makes it sound, but the SCC quotient is label-blind, so the bare reticulate is not faithful (Section 4.2). This suggests a graded hierarchy of increasingly faithful reticulates, refining the abstraction until it becomes reversible. The first step is concrete: we conjecture that replacing the SCC quotient by a *coinductive* quotient — unfolding each recursion once and identifying states up to labelled bisimulation — would yield a *refined reticulate* that is again a bounded lattice and is *faithful*, retaining the labels, polarity, and recursive continuity that end-identification and the SCC quotient forget, and would characterise Gay–Hole subtyping exactly — $S \leq_{GH} T$ iff the width-preserving embedding between the refined reticulates of S and T holds (supertype into subtype on branch, subtype into supertype on selection).

A Galois connection between an implementation's concrete state space and such a faithful reticulate would then yield a sound abstract interpretation for protocol conformance [7]. Our duality is structural — it identifies session types whose state spaces coincide via polarity erasure (Lemma 9). Richer semantic dualities, between method pairs (e.g., reader/writer, producer/consumer) or with directional payload tags, fit naturally since the construction is invariant under method renaming; [14]'s channel duality, viewed through this lens, configures the two endpoints as a 2-arm parallel composition with mutually dual arms (modulo $?/!$), and a formal translation including the $?/!$ half is open. Three grammar extensions preserve lattice structure and are the subject of companion work: first-class sequential composition, synchronous parallel, and replication.

The Reticulate Theorem suggests that session types are more structured than previously recognised: the lattice emerges from the syntax rather than being imposed by the theory.

Acknowledgements

The author thanks the anonymous reviewers of ICE 2026, whose reviews improved not only the paper but the theory itself, exposing gaps that sharpened the results during the rebuttal round. Any errors are the author's alone.

References

[1] Coşku Acay & Frank Pfenning (2017): *Intersections and Unions of Session Types*. In: *Proc. 8th Workshop on Intersection Types and Related Systems (ITRS'16)*, *Electronic Proceedings in Theoretical Computer Science* 242, pp. 4–19, doi:10.4204/EPTCS.242.3. arXiv:1702.02272.

[2] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos & Nobuko Yoshida (2016): *Behavioral Types in Programming Languages*. *Foundations and Trends in Programming Languages* 3(2–3), pp. 95–230, doi:10.1561/2500000031.

[3] Franco Barbanera & Ugo de'Liguoro (2015): *Sub-behaviour Relations for Session-Based Client/Server Systems*. *Mathematical Structures in Computer Science* 25(6), pp. 1339–1381, doi:10.1017/S0960129514000322.

[4] Garrett Birkhoff (1967): *Lattice Theory*, 3rd edition. *Colloquium Publications* 25, American Mathematical Society, doi:10.1090/coll/025.

[5] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In: *CONCUR 2010 – Concurrency Theory*, *Lecture Notes in Computer Science* 6269, Springer, pp. 222–236, doi:10.1007/978-3-642-15375-4_16.

[6] Giuseppe Castagna & Alain Frisch (2005): *A Gentle Introduction to Semantic Subtyping*. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP)*, ACM, pp. 198–208, doi:10.1145/1069774.1069793.

[7] Patrick Cousot & Radhia Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In: *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, ACM, pp. 238–252, doi:10.1145/512950.512973.

[8] Francesco Dagnino, Paola Giannini & Mariangiola Dezani-Ciancaglini (2023): *Deconfined Global Types for Asynchronous Sessions*. *Logical Methods in Computer Science* 19(1), doi:10.46298/lmcs-19(1:3)2023.

[9] Ornela Dardha & Simon J. Gay (2018): *A New Linear Logic for Deadlock-Free Session-Typed Processes*. In: *Foundations of Software Science and Computation Structures (FoSSaCS)*, Springer, pp. 91–109, doi:10.1007/978-3-319-89366-2_5.

[10] Ornela Dardha, Elena Giachino & Davide Sangiorgi (2017): *Session Types Revisited*. *Information and Computation* 256, pp. 253–286, doi:10.1016/j.ic.2017.06.002.

[11] B. A. Davey & H. A. Priestley (2002): *Introduction to Lattices and Order*, 2nd edition. Cambridge University Press, doi:10.1017/CBO9780511809088.

[12] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida & Sophia Drossopoulou (2006): *Session Types for Object-Oriented Languages*. In: *European Conference on Object-Oriented Programming (ECOOP)*, Springer, pp. 328–352, doi:10.1007/11785477_20.

[13] Simon J. Gay & Malcolm Hole (2005): *Subtyping for Session Types in the Pi Calculus*. Acta Informatica 42(2–3), pp. 191–225, doi:10.1007/s00236-005-0177-z.

[14] Simon J. Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert & Alexandre Zua Caldeira (2010): *Modular Session Types for Distributed Object-Oriented Programming*. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM, pp. 299–312, doi:10.1145/1706299.1706335.

[15] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: European Symposium on Programming (ESOP), Lecture Notes in Computer Science 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567.

[16] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty Asynchronous Session Types*. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM, pp. 273–284, doi:10.1145/1328438.1328472.

[17] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira & Gianluigi Zavattaro (2016): *Foundations of Session Types and Behavioural Contracts*. ACM Computing Surveys 49(1), pp. 3:1–3:36, doi:10.1145/2873052.

[18] Naoki Kobayashi (2006): *A New Type System for Deadlock-Free Processes*. In: CONCUR, Springer, pp. 233–247, doi:10.1007/11817949_16.

[19] Dimitrios Kouzapas, Ornela Dardha, Roly Perera & Simon J. Gay (2016): *Typechecking Protocols with Mungo and StMungo*. In: Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming (PPDP), ACM, pp. 146–159, doi:10.1145/2967973.2968595.

[20] Dimitrios Kouzapas, Ornela Dardha, Roly Perera & Simon J. Gay (2018): *Typechecking Protocols with Mungo and StMungo: A Session Type Toolchain for Java*. Science of Computer Programming 155, pp. 52–75, doi:10.1016/j.scico.2017.10.006.

[21] Robin Milner (1989): *Communication and Concurrency*. Prentice Hall.

[22] Leonardo de Moura & Sebastian Ullrich (2021): *The Lean 4 Theorem Prover and Programming Language*. In: Automated Deduction – CADE 28, Lecture Notes in Computer Science 12699, Springer, pp. 625–635, doi:10.1007/978-3-030-79876-5_37.

[23] Luca Padovani (2009): *Session Types at the Mirror*. In: Proc. 2nd Interaction and Concurrency Experience (ICE'09), Electronic Proceedings in Theoretical Computer Science 12, pp. 71–86, doi:10.4204/EPTCS.12.6.

[24] Luca Padovani (2011): *Session Types = Intersection Types + Union Types*. In: Proc. 5th Workshop on Intersection Types and Related Systems (ITRS'10), Electronic Proceedings in Theoretical Computer Science 45, pp. 71–89, doi:10.4204/EPTCS.45.6.

[25] Luca Padovani (2016): *Fair Subtyping for Multi-party Session Types*. Mathematical Structures in Computer Science 26(3), pp. 424–464, doi:10.1017/S096012951400022X.

[26] Davide Sangiorgi & David Walker (2001): *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, doi:10.1017/9781316134924.

[27] Alceste Scalas & Nobuko Yoshida (2019): *Less Is More: Multiparty Session Types Revisited*. Proceedings of the ACM on Programming Languages 3(POPL), pp. 30:1–30:29, doi:10.1145/3290343.

[28] Robert E. Strom & Shaula Yemini (1986): *Typestate: A Programming Language Concept for Enhancing Software Reliability*. IEEE Transactions on Software Engineering SE-12(1), pp. 157–171, doi:10.1109/TSE.1986.6312929.

[29] Robert Tarjan (1972): *Depth-First Search and Linear Graph Algorithms*. SIAM Journal on Computing 1(2), pp. 146–160, doi:10.1137/0201010.

[30] The mathlib Community (2020): *The Lean Mathematical Library*. Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP), pp. 367–381, doi:10.1145/3372885.3373824.

[31] Bernardo Toninho, Luís Caires & Frank Pfenning (2013): *Higher-Order Processes, Functions, and Sessions: A Monadic Integration*. In: *European Symposium on Programming (ESOP)*, *Lecture Notes in Computer Science* 7792, Springer, pp. 350–369, doi:10.1007/978-3-642-37036-6_20.

[32] Vasco T. Vasconcelos (2012): *Fundamentals of Session Types*. *Information and Computation* 217, pp. 52–70, doi:10.1016/j.ic.2012.05.002.

[33] Philip Wadler (2012): *Propositions as Sessions*. In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, ACM, pp. 273–286, doi:10.1145/2364527.2364568.

[34] Alexandre Zua Caldeira (2026): *Session Type State Spaces Form Lattices — companion artifact*, doi:10.5281/zenodo.21450198.

A Detailed Proofs

This appendix collects the manual proofs of the lemmas and theorems of Section 3, together with the structural argument for Proposition 18.

Proof of Proposition 7

Proof. By induction on the construction: each case introduces $q_\top$ and $q_\bot$ (End: single state; choice: fresh $q_\top$ plus shared $q_\bot$; recursion: back-edges preserve extrema in the quotient; parallel: componentwise extrema). □

Proof of Lemma 8

Proof. The state space $\mathscr{L}(\mathbf{end})$ consists of a single state q that is both initial and terminal: $q_\top = q_\bot = q$. The SCC quotient $\mathscr{L}(\mathbf{end})/\equiv$ is the one-element poset $\{[q]\}$, which satisfies all lattice axioms: $[q]$ is both top and bottom, and $[q] \wedge [q] = [q] \vee [q] = [q]$. □

Proof of Lemma 11

Proof. Since D is a proper upward-closed subset of L, $\bot \notin D$ (otherwise upward-closure would force $D = L$), so L' retains both extrema. For $a, b \in L \setminus D$: the meet $a \wedge_L b$ lies in $L \setminus D$ (the complement of an upward-closed set is downward-closed), so it is preserved. For the join: if $a \vee_L b \in L \setminus D$, it is preserved. If $a \vee_L b \in D$, then $a \vee_{L'} b = \top$: any non-$\top$ upper bound $u \in L \setminus D$ would satisfy $u \geq a \vee_L b \in D$, placing $u \in D$ since D is upward-closed — contradicting $u \in L \setminus D$. □

Proof of Lemma 12

Proof. Adding back-edges from each placeholder p_X to the initial state $q^S_\top$ makes these states mutually reachable with $q^S_\top$, hence they are collapsed into the same equivalence class $[q^S_\top]$. The set D of quotient classes absorbed into $[q^S_\top]$ is upward-closed: if $[s] \in D$ and $[s] \leq [r]$, then $r \twoheadrightarrow s$ (since $[r] \geq [s]$), $s \twoheadrightarrow q^S_\top$ (since $[s] \in D$), and $q^S_\top \twoheadrightarrow r$ (as initial state); so r and $q^S_\top$ are mutually reachable, giving $[r] \in D$. By the Termination clause, $[q_\bot]$ is reachable from $[q^S_\top]$ in $\mathscr{L}(\mu X.S)/\equiv$, so $[q_\bot] \notin D$; hence D is a *proper* upward-closed subset of the body quotient. By Lemma 11, collapsing D into a single top element yields a bounded lattice. □

Proof of Lemma 13

Proof. The construction connects the k named sub-state-spaces by redirecting placeholder transitions for each name X_j to $q_\top^{S_j}$. Non-recursive references (acyclic names) are effectively inlined; when non-reconvergent (the lemma's hypothesis) they preserve lattice structure, reconvergent sharing being the bounded-partial-lattice case of Section 3.3. Cyclic references create back-edges that merge states into SCCs, exactly as in the single-recursion case. The set of quotient classes absorbed into each SCC is upward-closed by the same argument as Lemma 12. Applying Lemma 11 iteratively (once per SCC created by cyclic references) yields a bounded lattice. The SCC quotient is determined by the reachability graph alone (Definition 6); independence from the absorption order is a confluence property mechanised as `NamedDefinitions.absorption_order_independent`. □

Proof of Lemma 14

Proof. By the parallel case of Definition 3 and the independence of the n branches (Termination ensures every branch reaches **end**; Parallel closedness ensures no shared variables), the product inherits SCC structure componentwise — an SCC in the product is a product of SCCs from each component:

$$\mathscr{L}(S_1 \parallel \cdots \parallel S_n)/{\equiv} \;\cong\; \prod_{i=1}^{n} (\mathscr{L}(S_i)/{\equiv_i}).$$

The product of finitely many bounded lattices is a bounded lattice (meets and joins are componentwise). See Figure 3 for an illustration with $n = 2$. □

Proof of Theorem 15

Proof. By structural induction on S, applying Lemmas 8, 10 and 12 to 14 case-by-case. Each case preserves the bounded lattice property, completing the induction. □

Proof of Proposition 18

Proof. By structural induction on S.

Base case ($S = \mathbf{end}$): $\mathit{dual}(\mathbf{end}) = \mathbf{end}$, so the state spaces are identical.

Choice ($S = \&\{m_1{:}S_1, \ldots, m_n{:}S_n\}$): $\mathit{dual}(S) = \oplus\{m_1{:}\mathit{dual}(S_1), \ldots, m_n{:}\mathit{dual}(S_n)\}$. Branch and selection produce identical state spaces (Lemma 9), so it suffices that the continuations agree: by induction $\mathscr{L}(S_i)/{\equiv} \cong \mathscr{L}(\mathit{dual}(S_i))/{\equiv}$ for each i, and the n sub-isomorphisms with the identity on $q_\top, q_\bot$ give the isomorphism of the whole choice state space.

Recursion ($S = \mu X.S'$): $\mathit{dual}(\mu X.S') = \mu X.\mathit{dual}(S')$. The recursion construction redirects placeholder transitions to the body's initial state. Since dual acts on the body's constructors (swapping & and $\oplus$) but preserves the placeholder and entry-state structure, the same back-edges are created in both $\mathscr{L}(\mu X.S')$ and $\mathscr{L}(\mu X.\mathit{dual}(S'))$, yielding isomorphic SCC quotients by induction on the body.

Parallel ($S = S_1 \parallel \cdots \parallel S_n$): dual distributes over parallel, so $\mathit{dual}(S) = \mathit{dual}(S_1) \parallel \cdots \parallel \mathit{dual}(S_n)$. By induction each factor is isomorphic; the product of isomorphic lattices is isomorphic.

Named definitions ($D = S$, $X_1 = S_1, \ldots, X_k = S_k$): as in the recursion case, *dual* acts on each equation body but preserves the name-reference structure, so the same redirections occur in both $\mathscr{L}(D)$ and $\mathscr{L}(\mathit{dual}(D))$; the SCC quotients are isomorphic by induction on each S_i. □