Definitions and Glossary

Session Types as Algebraic Reticulates

Alexandre Zua Caldeira
LASIGE/FCUL, University of Lisbon

February 25, 2026

Abstract

This document collects formal and informal definitions for all concepts in the *reticulate* project, organized by topic. It serves as a self-contained reference for the theory, algorithms, and tooling developed for the study of session types as algebraic reticulates.

Contents

1 Session Types

Definition 1.1 (Session Type)**.** A *session type* is a type that describes a communication protocol on an object: the legal sequences of method calls, branches, selections, and concurrent access patterns that a client may perform. A session type prescribes the *dynamic interface* of an object—the set of available methods changes as the protocol progresses.

Definition 1.2 (Constructor)**.** A *constructor* is a syntactic building block of the session type grammar. The six constructors are:

(i) **Branch** (&) — external choice,

(ii) **Selection** ($\oplus$) — internal choice,

(iii) **Parallel** ($\|$) — concurrent composition,

(iv) **Recursion** (μ) — recursive type,

(v) **Type variable** (X) — recursion reference,

(vi) **Termination** (**end**) — completed protocol.

Sequencing (.) is syntactic sugar for a single-method branch.

Definition 1.3 (Branch — External Choice)**.** A *branch* $\&\{m_1\colon S_1,\ \ldots,\ m_n\colon S_n\}$ offers methods $m_1, \ldots, m_n$ to the *client*, who chooses which to call. If the client calls method m_i, the protocol continues as S_i. The object must be prepared to handle any of the offered methods. In the state space, a branch creates a single state with n outgoing transitions.

Definition 1.4 (Selection — Internal Choice)**.** A *selection* $\oplus\{l_1\colon S_1,\ \ldots,\ l_n\colon S_n\}$ lets the *object* choose which label to produce. After a method returns, the label l_i determines the continuation S_i. The client must be prepared to handle any of the possible outcomes. In the state space, a selection creates a single state with n outgoing transitions (structurally identical to branch, but with different semantic intent).

Definition 1.5 (Parallel Composition)**.** A *parallel composition* $S_1 \parallel S_2$ forks the protocol into two concurrent sub-protocols. Methods from S_1 and S_2 may interleave freely. When both sub-protocols reach **end**, the fork-join completes and execution resumes sequentially. In the state space, parallel composition produces the *product* of the component state spaces. This is the key constructor that forces lattice structure.

Definition 1.6 (Recursion)**.** A *recursion* $\mu X.\, S$ binds a type variable X in the body S. When evaluation reaches X, it loops back to the beginning of the μ-binder. In the state space, recursion creates a back-edge (cycle) from the variable's occurrence to the body's entry state.

Definition 1.7 (Type Variable)**.** A *type variable* X is a reference to an enclosing recursion binder $\mu X.\, S$. It represents the point where the protocol loops back. A variable must be:

- **Bound**: enclosed by a corresponding μ-binder, and
- **Guarded**: preceded by at least one observable action (&, $\oplus$, or method call).

Definition 1.8 (Termination)**.** The constructor **end** represents a finished protocol: no further method calls are permitted. It corresponds to the bottom element ($\bot$) of the state space.

Definition 1.9 (Sequencing — Syntactic Sugar)**.** *Sequencing* $m\,.\,S$ desugars to $\&\{m\colon S\}$—a single-method branch. In a chain $a\,.\,b\,.\,c\,.\,\mathbf{end}$, each step represents a mandatory method call before proceeding to the next.

Definition 1.10 (Linear Ownership)**.** *Linear ownership* is the assumption, in standard session type systems, that exactly one client holds a reference to the object at any time. This precludes modeling concurrent access. The parallel constructor ($\|$) relaxes this assumption within its scope.

2 State Space

Definition 2.1 (State Space — Labeled Transition System)**.** Given a session type S, its *state space* $\mathcal{L}(S) = (Q,\ \Sigma,\ \delta,\ q_\top,\ q_\bot)$ consists of:

- Q: a finite set of *states* (integer identifiers),
- Σ: an alphabet of method names and selection labels,
- $\delta \subseteq Q \times \Sigma \times Q$: the *transition relation*,
- $q_\top$ (*top*): the initial protocol state,
- $q_\bot$ (*bottom*): the terminal state (**end**).

Each state represents a point in the protocol; each transition represents a method call or selection outcome.

Definition 2.2 (Top — $\top$)**.** The *top* element $q_\top$ is the initial state of the protocol—the entry point before any method has been called. In the reachability ordering, top is the *greatest* element: it can reach every other state.

Definition 2.3 (Bottom — $\bot$)**.** The *bottom* element $q_\bot$ is the terminal state of the protocol—the **end** state. In the reachability ordering, bottom is the *least* element: every other state can reach it (in well-formed types).

Definition 2.4 (Transition)**.** A *transition* is a triple (s, m, s') meaning: from state s, calling method m (or receiving label m) leads to state s'. Written $s \xrightarrow{m} s'$.

Definition 2.5 (Reachability)**.** State s_1 *reaches* state s_2 (written $s_1 \geq s_2$) if there is a directed path from s_1 to s_2 in the transition graph. This defines a preorder on states.

Definition 2.6 (Enabled Transitions)**.** For a state s, the set of *enabled transitions* is

$$\text{enabled}(s) = \{(m, s') \mid (s, m, s') \in \delta\}.$$

This determines the *dynamic interface* at state s—the methods available to the client.

3 Product Construction

Definition 3.1 (Product State Space)**.** Given two state spaces $\mathcal{L}_1$ and $\mathcal{L}_2$ (from $S_1 \parallel S_2$), their *product* is

$$\mathcal{L}(S_1 \parallel S_2) = \mathcal{L}_1 \times \mathcal{L}_2.$$

States are pairs (s_1, s_2) representing concurrent configurations: "path 1 is in state s_1, path 2 is in state s_2." Transitions advance one component at a time:

$$(s_1, s_2) \xrightarrow{a} (s_1', s_2) \qquad \text{if } s_1 \xrightarrow{a} s_1' \text{ in } \mathcal{L}_1, \tag{1}$$

$$(s_1, s_2) \xrightarrow{a} (s_1, s_2') \qquad \text{if } s_2 \xrightarrow{a} s_2' \text{ in } \mathcal{L}_2. \tag{2}$$

Top is $(\top_1, \top_2)$ (the fork point); bottom is $(\bot_1, \bot_2)$ (the join point).

Definition 3.2 (Componentwise Ordering)**.** The product is ordered componentwise:

$$(s_1, s_2) \leq (s_1', s_2') \quad \iff \quad s_1 \leq_1 s_1' \ \text{ and } \ s_2 \leq_2 s_2'.$$

This ordering inherits lattice structure from the components.

Definition 3.3 (Fork Point)**.** The *fork point* $(\top_1, \top_2)$ is the top element of a product state space. It represents the moment when the protocol splits into two concurrent paths.

Definition 3.4 (Join Point)**.** The *join point* $(\bot_1, \bot_2)$ is the bottom element of a product state space. It represents the moment when both concurrent paths have completed and execution resumes sequentially.

Definition 3.5 (Interleaving)**.** In the product construction, transitions from S_1 and S_2 may occur in any order. The set of all possible orderings of transitions from both components forms the set of valid *interleavings*. The product state space captures all interleavings as paths from top to bottom.

4 Order Theory

Definition 4.1 (Partially Ordered Set — Poset)**.** A *partially ordered set* (poset) is a set P equipped with a binary relation $\leq$ that is:

(i) **Reflexive**: $a \leq a$,

(ii) **Antisymmetric**: $a \leq b$ and $b \leq a$ implies $a = b$,

(iii) **Transitive**: $a \leq b$ and $b \leq c$ implies $a \leq c$.

The reachability relation on a state space (after SCC quotient) forms a poset.

Definition 4.2 (Lattice)**.** A *lattice* is a poset in which every pair of elements has both a *meet* (greatest lower bound) and a *join* (least upper bound). Equivalently, a *bounded lattice* has a top element $\top$ and bottom element $\bot$, and all pairwise meets and joins exist.

Definition 4.3 (Reticulate)**.** A *reticulate* is the lattice (or lattice-like poset) formed by the state space of a session type under the reachability ordering. This is the central concept of the thesis: session-type state spaces are *algebraic reticulates*—lattices arising from the algebraic structure of the type constructors.

Definition 4.4 (Meet — Greatest Lower Bound)**.** The *meet* of a and b, written $a \wedge b$, is the greatest element that is less than or equal to both:

$$a \wedge b = \max\{c \mid c \leq a \text{ and } c \leq b\}.$$

In a state space, the meet of two states is the "latest" state reachable from both—the point where the two execution paths most recently converge.

Definition 4.5 (Join — Least Upper Bound)**.** The *join* of a and b, written $a \vee b$, is the least element that is greater than or equal to both:

$$a \vee b = \min\{c \mid a \leq c \text{ and } b \leq c\}.$$

In a state space, the join of two states is the "earliest" state from which both can be reached—the latest common ancestor in the reachability order.

Definition 4.6 (Bounded Poset)**.** A *bounded poset* is a poset with both a greatest element (top, $\top$) and a least element (bottom, $\bot$). Every state-space poset is bounded: $\top$ is the initial state and $\bot$ is the **end** state.

Definition 4.7 (Upper Bound / Lower Bound)**.** An *upper bound* of a and b is any element c with $a \leq c$ and $b \leq c$. A *lower bound* is any element c with $c \leq a$ and $c \leq b$.

Definition 4.8 (Hasse Diagram)**.** A *Hasse diagram* is a graphical representation of a poset in which:

- Elements are drawn as nodes,
- If $a > b$ and there is no c with $a > c > b$ (i.e., a *covers* b), an edge is drawn from a to b,
- Greater elements are placed higher.

In our diagrams, $\top$ (initial state) is at the top and $\bot$ (**end**) is at the bottom. Transition labels are shown on edges.

5 Strongly Connected Components and Quotients

Definition 5.1 (Strongly Connected Component — SCC)**.** A *strongly connected component* (SCC) is a maximal set of states in which every state is reachable from every other state. SCCs arise from recursion: $\mu X.\ \&\ \{a\colon X,\ \mathit{done}\colon \mathbf{end}\}$ creates a cycle from the branch state back to itself.

Definition 5.2 (Quotient by SCCs)**.** The *quotient* operation collapses each SCC into a single node. If states s_1 and s_2 are in the same SCC (mutually reachable), they are identified. The result is a directed acyclic graph (DAG) called the *quotient DAG*.

This is necessary because cycles violate antisymmetry: if s_1 reaches s_2 and s_2 reaches s_1, then $s_1 \geq s_2$ and $s_2 \geq s_1$ but $s_1 \neq s_2$. Quotienting restores antisymmetry, yielding a proper poset.

Definition 5.3 (SCC Representative)**.** For each SCC, a canonical member (the smallest state ID in the component) used to represent the entire SCC in the quotient DAG.

Definition 5.4 (Tarjan's Algorithm)**.** *Tarjan's algorithm* computes all SCCs via a single depth-first traversal in $O(|Q| + |\delta|)$ time. The implementation in RETICULATE uses an iterative (non-recursive) variant to avoid stack overflow on deep graphs.

Definition 5.5 (Quotient DAG)**.** The *quotient DAG* is the directed acyclic graph obtained after collapsing SCCs. Edges in the quotient DAG represent transitions between different SCCs. Lattice properties are checked on this DAG.

6 Lattice Verification

Definition 6.1 (Lattice Check)**.** The *lattice check* is the algorithm that verifies whether a state space forms a lattice. It operates on the quotient DAG and checks four conditions:

(i) **Top reachable**: the top SCC reaches all other SCCs,

(ii) **Bottom reachable**: all SCCs reach the bottom SCC,

(iii) **All meets exist**: every pair of quotient nodes has a greatest lower bound,

(iv) **All joins exist**: every pair of quotient nodes has a least upper bound.

The state space is a lattice if and only if all four conditions hold.

Definition 6.2 (Counterexample)**.** When the lattice check fails, a *counterexample* is a pair of states (a, b) together with the failing property (`no_meet` or `no_join`). This identifies the specific structural defect in the state space.

Definition 6.3 (Lattice Result)**.** The output of the lattice check: a record containing `is_lattice`, `has_top`, `has_bottom`, `all_meets_exist`, `all_joins_exist`, `num_scc`, `counterexample`, and `scc_map`.

Theorem 6.4 (Product Lattice Theorem)**.** *If $\mathcal{L}_1$ and $\mathcal{L}_2$ are lattices, then $\mathcal{L}_1 \times \mathcal{L}_2$ is a lattice, with componentwise meets and joins:*

$$(s_1, s_2) \wedge (s_1', s_2') = (s_1 \wedge_1 s_1',\ s_2 \wedge_2 s_2'), \tag{3}$$

$$(s_1, s_2) \vee (s_1', s_2') = (s_1 \vee_1 s_1',\ s_2 \vee_2 s_2'). \tag{4}$$

This is the fundamental theorem connecting parallel composition to lattice structure.

7 Well-Formedness and Termination

Definition 7.1 (Termination)**.** A recursive session type $\mu X.\, S$ is *terminating* if the body S has at least one syntactic path from its root to a leaf that is not X. This ensures the recursion can eventually exit, reaching **end**.

Example 7.2 (Terminating vs. Non-Terminating).
Terminating: $\mu X.\ \&\{\mathit{read}\colon X,\ \mathit{done}\colon \mathbf{end}\}$ — the *done* branch exits.
Non-terminating: $\mu X.\ \&\{\mathit{loop}\colon X\}$ — all paths loop back.

Definition 7.3 (Exit Path)**.** An *exit path* is a syntactic path through the AST from the root of a recursive body to **end** (or to a variable bound by an *outer* recursion) that does not pass through the forbidden variable X. The existence of an exit path is the termination criterion.

Definition 7.4 (WF-Par — Well-Formedness of Parallel)**.** For each occurrence of $S_1 \parallel S_2$, three conditions must hold:

(i) **Termination**: both S_1 and S_2 are terminating. This ensures the join point $(\mathbf{end}, \mathbf{end})$ is always reachable.

(ii) **No cross-branch variables**: no recursion variable bound by μ in S_1 occurs free in S_2, and vice versa. This ensures the branches can be built as independent state spaces.

(iii) **No nested parallel**: neither S_1 nor S_2 contains a $\parallel$ constructor. Nested parallelism is deferred to a future version.

Definition 7.5 (Guardedness)**.** A recursion variable X is *guarded* in S if every occurrence of X is preceded by at least one observable action (branch, selection, or method call). This prevents non-productive definitions like $\mu X.\, X$. The $\parallel$ constructor does *not* act as a guard.

Definition 7.6 (Free Variables)**.** The *free variables* of a session type are the type variables that occur in it but are not bound by any enclosing μ. Used in the cross-branch variable check for WF-Par.

Definition 7.7 (Bound Variables)**.** The *bound variables* of a session type are the type variables introduced by μ-binders within it. Used in the cross-branch variable check for WF-Par.

8 Morphisms

Definition 8.1 (Morphism Between State Spaces)**.** A *morphism* $f\colon Q_S \to Q_T$ is a mapping between the state sets of two state spaces. Morphisms classify the structural relationship between protocols.

Definition 8.2 (Order-Preserving)**.** A map f is *order-preserving* if

$$s_1 \geq s_2 \implies f(s_1) \geq f(s_2).$$

Reachability in the source implies reachability in the target.

Definition 8.3 (Order-Reflecting)**.** A map f is *order-reflecting* if

$$f(s_1) \geq f(s_2) \implies s_1 \geq s_2.$$

Reachability in the target implies reachability in the source.

Definition 8.4 (Morphism Hierarchy)**.** Four levels of morphism, ordered by strength:

Kind	**Preserving**	**Reflecting**	**Injective**	**Surjective**
Homomorphism	✓			
Projection	✓			✓
Embedding	✓	✓	✓	
Isomorphism	✓	✓	✓	✓

Every isomorphism is an embedding; every embedding is a homomorphism; every projection is a homomorphism.

Definition 8.5 (Isomorphism)**.** An *isomorphism* is a bijective, order-preserving, order-reflecting map. Two state spaces are isomorphic iff they have the same structure up to renaming of states. Isomorphism means the protocols are *equivalent*.

Definition 8.6 (Embedding)**.** An *embedding* is an injective, order-preserving, order-reflecting map. The source "embeds" faithfully into the target: all structure is preserved, but the target may have additional states. Embedding means one protocol is a *sub-protocol* of another.

Definition 8.7 (Projection)**.** A *projection* is a surjective, order-preserving map. The source "projects onto" the target: every target state has a preimage, but order-reflection is not required. Projection means one protocol *abstracts* another.

Definition 8.8 (Homomorphism)**.** A *homomorphism* is an order-preserving map (the weakest morphism). Structure is preserved in one direction only.

Definition 8.9 (Galois Connection)**.** A *Galois connection* is a pair of maps (α, γ) between state spaces S (concrete) and T (abstract) satisfying the *adjunction condition*:

$$\alpha(x) \leq y \iff x \leq \gamma(y) \qquad \text{for all } x \in Q_S,\ y \in Q_T.$$

Here α is the *abstraction* and γ is the *concretization*. This generalizes the Cousot–Cousot framework of abstract interpretation to session type state spaces.

9 Thread Safety

Remark 9.1. The definitions in this section pertain to BICA Reborn, the planned Java annotation-based session type checker. They are included for completeness.

Definition 9.2 (Concurrency Level)**.** A *concurrency level* classifies each method's thread-safety properties:

Level	Symbol	Meaning
EXCLUSIVE	ε	Requires sole access (default)
READ_ONLY	ρ	Safe with other readers
SYNC	σ	Java `synchronized`—serialized access
SHARED	$\star$	Programmer asserts full thread safety

These levels form a lattice: EXCLUSIVE $<$ {READ_ONLY, SYNC} $<$ SHARED.

Definition 9.3 (Concurrent Compatibility)**.** Two methods m_1 and m_2 are *concurrently compatible* if their concurrency levels permit safe concurrent execution. The rule: any pair involving EXCLUSIVE is unsafe; all other pairs are safe. Formally:

$$\text{compatible}(\ell_1, \ell_2) \iff \ell_1 \wedge \ell_2 \neq \text{EXCLUSIVE}$$

where $\wedge$ is the meet in the concurrency-level lattice.

Definition 9.4 (Trust Boundary)**.** When a method is annotated `@Shared`, the programmer asserts thread safety through mechanisms the tool cannot verify (explicit locks, atomics, lock-free algorithms). These methods are *trust boundaries*: the tool verifies everything else and clearly reports where programmer judgment is required.

10 AST and Parser

Definition 10.1 (Abstract Syntax Tree — AST)**.** The *abstract syntax tree* is the tree representation of a parsed session type. Seven node types (implemented as frozen dataclasses):

(i) **End**: terminal node,

(ii) **Var**($name$): type variable reference,

(iii) **Branch**($choices$): external choice, $choices = ((m_1, S_1), \ldots, (m_n, S_n))$,

(iv) **Select**($choices$): internal choice, same structure as Branch,

(v) **Parallel**($left, right$): fork into two sub-protocols,

(vi) **Rec**($var, body$): recursion binder,

(vii) **Sequence**($left, right$): sequencing (desugared from $m \,.\, S$).

All nodes are immutable and hashable.

Definition 10.2 (Parser)**.** The *parser* is a recursive-descent parser that converts a session type string into an AST. It supports both ASCII (`||`, `+`, `rec`) and Unicode ($\|$, $\oplus$, μ) syntax.

Definition 10.3 (Pretty-Printer)**.** The *pretty-printer* converts an AST back into a human-readable string representation. Used for display in the CLI, web demo, and diagnostics.

Definition 10.4 (Desugaring)**.** *Desugaring* is the transformation of syntactic sugar into core syntax. The primary desugaring: $m \,.\, S \longmapsto$ **Branch**$(((m, S),))$—a single-method branch.

11 Visualization

Definition 11.1 (DOT Source)**.** A *DOT source* is a string in the Graphviz DOT language describing the Hasse diagram of a state space. Generated by `dot_source()` using only the Python standard library (no external dependencies beyond Graphviz for rendering).

Definition 11.2 (Hasse Diagram — Rendering)**.** A graphical rendering of a state space as a directed graph:

- Nodes represent states, labeled with the session type at that point,
- Edges represent transitions, labeled with method/selection names,
- Top ($\top$) is pinned to the top of the layout,
- Bottom ($\bot$) is pinned to the bottom,
- Top is colored blue; bottom is colored green,
- Counterexample states (if any) are highlighted in red.

Definition 11.3 (Counterexample Highlighting)**.** When the lattice check fails, the pair of states forming the counterexample is drawn with red borders and thicker outlines in the Hasse diagram, enabling visual identification of the structural defect.

12 Tooling

Definition 12.1 (Reticulate)**.** RETICULATE is the Python library and CLI tool that implements the full pipeline:

parse $\rightarrow$ build state space $\rightarrow$ check lattice $\rightarrow$ check termination $\rightarrow$ check WF-Par $\rightarrow$ classify morphisms

Named after the lattice structure ("reticulate" = net-like, lattice-like).

Definition 12.2 (BICA Reborn)**.** BICA REBORN is the planned Java annotation-based session type checker for objects. Successor to the original BICA (2009). It will use RETICULATE as the lattice verification back-end. Key novelty: the $\|$ constructor with thread safety checking.

Definition 12.3 (Pipeline)**.** The analysis *pipeline* is the sequence of transformations applied to a session type string:

$$\text{input} \rightarrow \text{parse} \rightarrow \text{build state space} \rightarrow \begin{cases} \text{check lattice} \\ \text{check termination} \\ \text{check WF-Par} \end{cases} \rightarrow \text{render Hasse diagram} \rightarrow \text{return result.}$$

The pipeline itself can be described as a session type (see `docs/self-reference.md`).

Definition 12.4 (Benchmark Protocol)**.** A *benchmark protocol* is a real-world or classic protocol expressed as a session type, with expected metrics (state count, transition count, SCC count). The 17 benchmarks include: Java Iterator, File Object, SMTP, HTTP, OAuth 2.0, Two-Buyer, MCP, A2A, File Channel, ATM, Reentrant Lock, WebSocket, DB Transaction, Pub/Sub, DNS Resolver, Reticulate Pipeline, and GitHub CI Workflow.