The Session Abstract Machine

Luís Caires[1] and Bernardo Toninho[2(✉)]

[1] Técnico Lisboa / INESC-ID, Lisbon, Portugal
luis.caires@tecnico.ulisboa.pt
[2] NOVA FCT / NOVA LINCS, Costa da Caparica, Portugal
btoninho@fct.unl.pt

Abstract. We build on a fine-grained analysis of session-based interaction as provided by the linear logic typing disciplines to introduce the SAM, an abstract machine for mechanically executing session-typed processes. A remarkable feature of the SAM's design is its ability to naturally segregate and coordinate sequential with concurrent session behaviours. In particular, implicitly sequential parts of session programs may be efficiently executed by deterministic sequential application of SAM transitions, amenable to compilation, and without concurrent synchronisation mechanisms. We provide an intuitive discussion of the SAM structure and its underlying design, and state and prove its correctness for executing programs in a session calculus corresponding to full classical linear logic CLL. We also discuss extensions and applications of the SAM to the execution of linear and session-based programming languages.

Keywords: Abstract machine · Session Types · Linear Logic

1 Introduction

In this work, we build on the linear logic based foundation for session types [13, 15, 72] to construct SAM, an abstract machine specially designed for executing session processes typed by (classical) linear logic CLL. Although motivated by the session type discipline, which originally emerged in the realm of concurrency and distribution [31, 33, 28, 34], a basic motivation for designing the SAM was to provide an efficient deterministic execution model for the implicitly sequential session-typed program idioms that often proliferate in concurrent session-based programming. It is well-known that in a world of fine-grained concurrency, building on many process-based encodings of concepts such as (abstract) data types, functions, continuations, and effects [49, 70, 65, 66, 10, 68, 54], large parts of the code turn out to be inherently sequential, further justifying the foundational and practical relevance of our results. A remarkable feature of the SAM's design is therefore its potential to efficiently coordinate sequential with full-fledged concurrent behaviours in session-based programming.

Leveraging early work relating linear logic with the semantics of linear and concurrent computation [1, 6, 2], the proposition-as-types (PaT) interpretation [73] of linear logic proofs as a form of well-behaved session-typed nominal calculus has motivated many developments since its inception [12, 5, 68, 67]. We believe that, much how the λ-calculus is deemed a canonical typed model for functional (sequential) computation with pure values, the session calculus can be accepted as a fairly canonical typed model for stateful concurrent computation with linear resources, well-rooted in the trunk of "classical" Type The-

S. Weirich (Ed.): ESOP 2024, LNCS 14576, pp. 206–235, 2024.
https://doi.org/10.1007/978-3-031-57262-3_9

ory. The PaT interpretation of session processes also establishes a bridge between more classical theories of computation and process algebra via logic. It also reinstates Robin Milner's view of computation as interaction [48], "data-as-processes" [49] and "functions-as-processes" [47], now in the setting of a tightly typed world, based on linear logic, where types may statically ensure key properties like deadlock-freedom, termination, and correct resource usage in stateful programs. Session calculi are motivating novel programming language design, bringing up new insights on typeful programming [18] with linear and behavioral types, e.g., [24, 61, 20, 5]. Most systems of typed session calculi have been formulated in process algebraic form [31, 33, 28], or on top of concurrent λ-calculi with an extra layer of communication channels (e.g., [29]), logically inspired systems such as the those discussed in this paper (e.g., [13, 15, 72, 23, 39, 59, 27, 61]) are defined by a logical proof / type system where proof rules are seen as witnesses for the typing of process terms, proofs are read as processes, structural equivalence is proof conversion and computation corresponds to cut reduction. These formulations provide a fundamental semantic foundation to study the model's expressiveness and meta-theory, but of course do not directly support the concrete implementation of programming languages based on them.

Although several programming language implementations of nominal calculi based languages have been proposed for some time (e.g. [57]), with some introducing abstract machines as the underlying technology (e.g., [69, 46]), we are not aware of any prior design proposal for an abstract machine for reducing session processes exploiting deep properties of a source session calculus, as e.g., the CAM [21] the LAM [41], or the KM [40], which also explore the Curry-Howard correspondences, may reclaim to be, respectively for call-by-value cartesian-closed structures, linear logic, and the call-by-name λ-calculus.

The SAM reduction strategy explores a form of "asynchronous" interaction that essentially expresses that, for processes typed by the logical discipline, sessions are always pairwise causally independent, in the sense that immediate communication on some session is never blocked by communication on a different session. This property is captured syntactically by prefix commutation equations, valid commuting conversions in the underlying logic: adding equations for such laws explicitly to process structural congruence keeps observational equivalence of CLL processes untouched [53]. Combined with insights related to focalisation and polarisation in linear logic [4, 56, 44], we realize that all communication in any session may be operationally structured as the exchange of bundles of positive actions from sender to receiver, where the roles sender/receiver flip whenever the session type swaps polarity. Communication may then be mediated by message buffers, first filled up by the sender ("write-biased" scheduling), and at a later time emptied by the receiver. Building on these observations and on key properties of linear logic proofs leveraged in well-known purely structural proofs of progress [13, 15, 61], we identify a sequential and deterministic reduction strategy for CLL typed processes, based on a form of co-routining where continuations are associated to session queues, and "context switching" occurs whenever polarity flips. That such strategy works at all, preserving all the required correctness

properties of the CLL language does not seem immediately obvious, given that each processes may sequentially perform multiple actions on many different sessions, meaning that multiple context switches must be interleaved. The bulk of our paper is then devoted to establishing all such properties in a precise technical sense. We believe that the SAM may provide a principled foundation for safe execution environments for programming languages combining functional, imperative and concurrent idioms based on session and linear types, as witnessed in practice for Rust [37], (Linear) Haskell [45, 8, 38], Move [9], and in research languages [60, 36, 24]. To further substantiate these views we have developed an implementation of the SAM, integrated in a language for realistic session-based shared-state programs [17].

Outline and Contributions. In Section 2 we briefly review the session-typed calculus CLL, which exactly corresponds to (classical) Linear Logic with mix. In Section 3 we discuss the motivation and design principles of the core SAM, gradually presenting its structure for the language fragment corresponding to session types without the exponentials, which will be introduced later. Even if the core SAM structure and transition rules are fairly simple, the proofs of correctness are more technically involved, and require progressive build up. Therefore, we first bridge between CLL and SAM via a intermediate logical language CLLB, introducing explicit queues in cuts, presented in Section 4. We show preservation (Theorem 4.1) and progress (Theorem 4.2) for CLLB, and prove that there is two way simulation between CLLB and CLL via a strong operational correspondence (Theorem 4.3). Given this correspondence, in Section 5 we state and prove the adequacy of the SAM for executing CLL processes, showing soundness wrt. CLLB (Theorem 5.1) and CLL (Theorem 5.2), and progress / deadlock absence (Theorem 5.3). In Section 6 modularly extend the previous results to the exponentials and mix, and revise the core SAM by introducing explicit environments, stating the associated adequacy results (Theorem 6.1 and Theorem 6.2). We also discuss how to accommodate concurrency, and other extensions in the SAM. We conclude by a discussion of related work and additional remarks. Additional definitions and proofs can be found in the companion technical report [16].

2 Background on CLL, the core language and type system

We start by revisiting the language and type system of CLL, and its operational semantics. The system is based on a PaT interpretation of classical linear logic (we follow the presentations of [15, 11, 60]).

Definition 2.1 (Types). *Types A, B are defined by*

$$A, B ::= \mathbf{1} \mid \bot \mid A \parr B \mid A \otimes B \mid \&_{\ell \in L} A_\ell \mid \oplus_{\ell \in L} A_\ell \mid !A \mid ?A$$

Types comprise of the units ($\mathbf{1}$, $\bot$), multiplicatives ($\otimes$, $\parr$), additives ($\oplus_{\ell \in L} A_\ell$, $\&_{\ell \in L} A_\ell$) and exponentials (!, ?). We adopt here a labeled version of the additives, where the linear logic sum type $A_{\#\mathsf{inl}} \oplus A_{\#\mathsf{inr}}$ is defined by $\oplus_{\ell \in \{\#\mathsf{inl}, \#\mathsf{inr}\}} A_\ell$. The *positive types* are $\mathbf{1}$, $\otimes$, $\oplus$, and !, while the *negative types* are $\bot$, $\parr$, $\&$ and ?. We abbreviate $\overline{A} \parr B$ by $A \multimap B$. We write A^+ (resp. A^-) to assert that A is a

positive (resp. negative) type. Type *duality* $\overline{A}$ corresponds to negation:

$$\overline{\mathbf{1}} = \bot \qquad \overline{A \otimes B} = \overline{A} \parr \overline{B} \qquad \overline{\oplus_{\ell \in L} A_\ell} = \&_{\ell \in L} \overline{A}_\ell \qquad \overline{!A} = ?\overline{B}$$

Duality captures the symmetry of behaviour in binary process interaction, as manifest in the cut rule.

Definition 2.2 (Processes). *The syntax of processes P, Q is given by:*

$$\begin{aligned} P, Q ::= {} & 0 \mid P \parallel Q \mid \mathsf{fwd}\ x\ y \mid \mathsf{cut}\ \{P \mid x{:}A \mid Q\} \mid \mathsf{close}\ x \mid \mathsf{wait}\ x; P \\ \mid {} & \mathsf{case}\ x\ \{\mid \#\ell \in L{:}P_\ell\} \mid \#\mathsf{inr}\ x; P \mid \mathsf{send}\ x(y.P); Q \mid \mathsf{recv}\ x(z); P \\ \mid {} & !x(y); P \mid ?x; P \mid \mathsf{cut!}\ \{y.P \mid !x : A \mid Q\} \mid \mathsf{call}\ x(z); Q \end{aligned}$$

Typing judgements have the form $P \vdash \Delta; \Gamma$, where P is a process and the *typing context* $\Delta; \Gamma$ is dyadic [4, 7, 55, 13]: both Δ and Γ assign types to names, the context Δ is handled linearly (no implicit contraction or weakening) while the exponential context Γ is unrestricted. The type system exactly corresponds, via a propositions-as-types correspondence, to the canonical proof system of Classical Linear Logic with Mix. When a cut type annotation is easily inferred, we may omit it and write $\mathsf{cut}\ \{P \mid x \mid Q\}$. The typing rules of CLL are given in Fig. 1.

The process 0 denotes the inactive process, typed in the empty linear context (rule [T0]). $P \parallel Q$ denotes independent parallel composition of processes P and Q (rule [Tmix]), whereas $\mathsf{cut}\ \{P \mid x{:}A \mid Q\}$ denotes interfering parallel composition of P and Q, where P and Q share exactly one channel name x, typed as A in P and $\overline{A}$ in Q (rule [Tcut]). The construct $\mathsf{fwd}\ x\ y$ captures forwarding between dually typed names x and y (rule [Tfwd]), which operationally consists in (globally) renaming x for y.

Processes $\mathsf{close}\ x$ and $\mathsf{wait}\ x; P$ denote session termination and the dual action of waiting for session termination, respectively (rules [T$\mathbf{1}$] and [T$\bot$]). The constructs $\mathsf{case}\ x\ \{\mid \#\ell \in L{:}P_\ell\}$ and $\#\mathsf{l}\ x; P$ denote label input and output, respectively, where the input construct pattern matches on the received label to select the process continuation that is to run. Process $\mathsf{send}\ x(y.P_1); P_2$ and $\mathsf{recv}\ x(z); Q$ codify the output of (fresh) name y on channel x and the corresponding input action, where the received name will be substituted for z in Q (rules [T$\otimes$] and [T$\parr$]). Typing ensures that the names used in P_1 and P_2 are disjoint.

Processes $!x(y); P$, $?x; Q$ and $\mathsf{call}\ x(z); Q$ embody replicated servers and client processes. Process $!x(y); P$ consists of a process that waits for inputs on x, spawning a replica of P (depending on no linear sessions – rule [T!]). Process $?x; Q$ and $\mathsf{call}\ x(z); Q$ allow for replicated servers to be activated and subsequently used as (fresh) linear sessions (rules [T?] and [Tcall]). Composition of exponentials is achieved by the $\mathsf{cut!}\ \{y.P \mid !x : A \mid Q\}$ process, where P cannot depend on linear sessions and so may be safely replicated.

We call *action* any process that is either a forwarder or realizes an introduction rule, and denote by $\mathcal{A}$ the set of all actions, by $\mathcal{A}(x)$ the set of action with subject x (the subject of an action is the channel name in which it interacts [49]). An action is deemed *positive* (resp. *negative*) if its associated type is positive (resp. negative) in the sense of *focusing*. The set of positive (resp.

$$\frac{}{\mathsf{0} \vdash \emptyset; \Gamma}\ [\mathrm{T0}] \qquad \frac{P \vdash \Delta'; \Gamma \quad Q \vdash \Delta; \Gamma}{P \parallel Q \ \vdash \Delta', \Delta; \Gamma}\ [\mathrm{Tmix}]$$

$$\frac{}{\mathsf{fwd}\ x\ y \ \vdash x : \overline{A}, y : A; \Gamma}\ [\mathrm{Tfwd}] \qquad \frac{P \vdash \Delta', x : A; \Gamma \quad Q \vdash \Delta, x : \overline{A}; \Gamma}{\mathsf{cut}\ \{P\ |x : A|\ Q\} \ \vdash \Delta', \Delta; \Gamma}\ [\mathrm{Tcut}]$$

$$\frac{}{\mathsf{close}\ x \vdash x : \mathbf{1}; \Gamma}\ [\mathrm{T}\mathbf{1}] \qquad \frac{Q \vdash \Delta; \Gamma}{\mathsf{wait}\ x; Q \ \vdash \Delta, x : \bot; \Gamma}\ [\mathrm{T}\bot]$$

$$\frac{P_\ell \vdash \Delta, x : A_\ell; \Gamma \quad (all\ \ell \in L)}{\mathsf{case}\ x\ \{|\#\ell \in L : P_\ell\} \ \vdash \Delta, x : \&_{\ell \in L} A_\ell; \Gamma}\ [\mathrm{T}\&] \qquad \frac{Q \vdash \Delta', x : A_{\#\mathsf{l}}; \Gamma \quad \#\mathsf{l} \in L}{\#\mathsf{l}\ x; Q \ \vdash \Delta', x : \oplus_{\ell \in L} A_\ell; \Gamma}\ [\mathrm{T}\oplus]$$

$$\frac{P_1 \vdash \Delta_1, y : A; \Gamma \quad P_2 \vdash \Delta_2, x : B; \Gamma}{\mathsf{send}\ x(y.P_1); P_2 \ \vdash \Delta_1, \Delta_2, x : A \otimes B; \Gamma}\ [\mathrm{T}\otimes]$$

$$\frac{Q \vdash \Delta, z : A, x : B; \Gamma}{\mathsf{recv}\ x(z); Q \vdash \Delta, x : A \parr B; \Gamma}\ [\mathrm{T}\parr]$$

$$\frac{P \vdash y : A; \Gamma}{!x(y); P \ \vdash x :!A; \Gamma}\ [\mathrm{T}!] \qquad \frac{Q \vdash \Delta; \Gamma, x : A}{?x; Q \ \vdash \Delta, x :?A; \Gamma}\ [\mathrm{T}?]$$

$$\frac{P \vdash y : A; \Gamma \quad Q \vdash \Delta; \Gamma, x : \overline{A}}{\mathsf{cut!}\ \{y.P\ |!x : A|\ Q\} \ \vdash \Delta; \Gamma}\ [\mathrm{Tcut!}] \qquad \frac{Q \vdash \Delta, z : A; \Gamma, x : A}{\mathsf{call}\ x(z); Q \ \vdash \Delta; \Gamma, x : A}\ [\mathrm{Tcall}]$$

Fig. 1: Typing Rules of CLL.

negative) actions is denoted by $\mathcal{A}^+$ (resp. $\mathcal{A}^-$). We sometimes use, e.g., $\mathcal{A}$ or $\mathcal{A}^+(x)$ to denote a process in the set. The CLL operational semantics is given by a *structural congruence* relation $\equiv$ that captures static identities on processes, corresponding to commuting conversions in the logic, and a *reduction* relation $\rightarrow$ that captures process interaction, and corresponds to cut-elimination steps.

Definition 2.3 ($P \equiv Q$). *Structural congruence $\equiv$ is the least congruence on processes closed under α-conversion and the $\equiv$-rules in Fig. 2.*

The definition of $\equiv$ reflects expected static laws, along the lines of the structural congruences / conversions in [13, 71]. The binary operators forwarder, cut, and mix are commutative. The set of processes modulo $\equiv$ is a commutative monoid with operation the parallel composition ($- \parallel -$) and identity given by inaction $\mathsf{0}$ ([par]). Any static constructs commute, as expressed by the laws [CM]-[C!sC!]. The unrestricted cut distributes over all the static constructs by law [C*], where $- |*| -$ stands for either a mix, linear or unrestricted cut. The laws [C+*] and [C+] denote sound proof equivalences in linear logic and bring explicit the independence of linear actions (noted $a(x)$), in different sessions x [53]. These conversions are not required to obtain deadlock freedom. However, they are necessary for full cut elimination (e.g., see [71]), and expose more redexes, thus more non-determinism in the choice of possible reductions. Perhaps surprisingly,

$$\mathsf{fwd}\ x\ y\ \equiv \mathsf{fwd}\ y\ x \qquad [\text{fwd}]$$

$$\mathsf{cut}\ \{P\ |x : A|\ Q\}\ \equiv \mathsf{cut}\ \{Q\ |x : \overline{A}|\ P\} \qquad [\text{com}]$$

$$P\ ||\ 0\ \equiv P \quad P\ ||\ Q \equiv Q\ ||\ P \quad P\ ||\ (Q\ ||\ R) \equiv (P\ ||\ Q)\ ||\ R \qquad [\text{par}]$$

$$\mathsf{cut}\ \{P\ |x|\ (Q\ ||\ R)\} \equiv (\mathsf{cut}\ \{P\ |x|\ Q\})\ ||\ R \qquad [\text{CM}]$$

$$\mathsf{cut}\ \{P\ |x|\ (\mathsf{cut}\ \{Q\ |y|\ R\})\} \equiv \mathsf{cut}\ \{(\mathsf{cut}\ \{P\ |x|\ Q\})\ |y|\ R\} \qquad [\text{CC}]$$

$$\mathsf{cut}\ \{P\ |z|\ (\mathsf{cut!}\ \{y.Q\ |!x|\ R\})\} \equiv \mathsf{cut!}\ \{y.Q\ |!x|\ (\mathsf{cut}\ \{P\ |z|\ R\})\} \qquad [\text{CC!}]$$

$$\mathsf{cut!}\ \{y.Q\ |!x|\ (P\ ||\ R)\} \equiv P\ ||\ (\mathsf{cut!}\ \{y.Q\ |!x|\ R\}) \qquad [\text{C!M}]$$

$$\mathsf{cut!}\ \{y.P\ |!x|\ (\mathsf{cut!}\ \{w.Q\ |!z|\ R\})\} \equiv \mathsf{cut!}\ \{w.Q\ |!z|\ (\mathsf{cut!}\ \{y.P\ |!x|\ R\})\} \qquad [\text{C!C!}]$$

$$\mathsf{cut!}\ \{y.P\ |!x|\ (Q\ |*|\ R)\} \equiv \mathsf{cut!}\ \{y.P\ |!x|\ Q\}\ |*|\ \mathsf{cut!}\ \{y.P\ |!x|\ R\} \qquad [\text{C!*}]$$

$$a(x); Q\ |*|\ R\ \equiv a(x); (Q\ |*|\ R) \qquad [\text{C+*}]$$

$$a_1(x); a_2(y); P\ \equiv a_2(y); a_1(x); P \qquad [\text{C}i]$$

Provisos: in [CM] $x \in \mathsf{fn}(Q)$; in [CC] $x, y \in \mathsf{fn}(Q)$; in [CC!], [C!M] $x \notin \mathsf{fn}(P)$; in [C!C!], $x \notin \mathsf{fn}(Q)$ and $z \notin \mathsf{fn}(P)$. In [Ci], $x \neq y$ and $bn(a_1(x)) \cap bn(a_2(y)) = \emptyset$

Fig. 2: Structural congruence $P \equiv Q$.

$$\mathsf{cut}\ \{\mathsf{fwd}\ x\ y\ |y|\ P\} \to \{x/y\}P \qquad [\text{fwd}]$$

$$\mathsf{cut}\ \{\mathsf{close}\ x\ |x|\ \mathsf{wait}\ x; P\} \to P \qquad [\mathbf{1}\bot]$$

$$\mathsf{cut}\ \{\mathsf{send}\ x(y.P); Q\ |x|\ \mathsf{recv}\ x(z); R\} \to Q\ |x|\ (P\ |y|\ \{y/z\}R) \qquad [\otimes\parr]$$

$$\mathsf{cut}\ \{\mathsf{case}\ x\ \{|\#\ell \in L{:}P_{\#\ell}\}\ |x|\ \#\mathsf{l}\ x; R\} \to \mathsf{cut}\ \{P_{\#\mathsf{l}}\ |x|\ R\} \qquad [\&\oplus_l]$$

$$\mathsf{cut}\ \{!x(y); P\ |x|\ ?x; Q\} \to \mathsf{cut!}\ \{y.P\ |!x|\ Q\} \qquad [!?]$$

$$\mathsf{cut!}\ \{y.P\ |!x|\ \mathsf{call}\ x(z); Q\} \to \mathsf{cut}\ \{\{z/y\}P\ |z|\ (\mathsf{cut!}\ \{y.P\ |!x|\ Q\})\} \qquad [\text{call}]$$

Fig. 3: Reduction $P \to Q$.

this extra flexibility is important to allow the deterministic sequential evaluation strategy for CLL programs adopted by the SAM to be expressed.

Definition 2.4 (Reduction $\to$). *Reduction $\to$ is defined by the rules of Fig. 3.*

We denote by $\Rightarrow$ the reflexive-transitive closure of $\to$. Reduction includes the set of principal cut conversions, i.e. the redexes for each pair of interacting constructs. It is closed by structural congruence ([$\equiv$]), in rule [cong] we consider that $\mathcal{C}$ is a static context, i.e. a process context in which the single hole is covered only by the static constructs mix or cut. The forwarding behaviour is implemented by name substitution [fwd] [14]. All the other reductions act on a principal cut between two dual actions, and eliminate it on behalf of cuts involving their subprocesses. CLL satisfies basic safety properties [13] listed below, and also confluence, and termination [60, 61]. In particular we have:

Theorem 2.1 (Type Preservation). *Let $P \vdash \Delta; \Gamma$. (1) If $P \equiv Q$, then $Q \vdash \Delta; \Gamma$. (2) If $P \to Q$, then $Q \vdash \Delta; \Gamma$.*

A process P is *live* if and only if $P = \mathcal{C}[Q]$, for some static context $\mathcal{C}$ (the hole lies within the scope of static constructs mix and cut) and Q is an active process (a process with a topmost action prefix).

Theorem 2.2 (Progress). *Let $P \vdash \emptyset; \emptyset$ be live. Then $P \to Q$ for some Q.*

3 A Core Session Abstract Machine

In this section we develop the key insights that guide the construction of our session abstract machine (SAM) and introduce its operational rules in an incremental fashion. We omit the linear logic exponentials for the sake of clarity of presentation, postponing their discussion for Section 6.

One of the main observations that drives the design of the SAM is the nature of proof dynamics in (classical) linear logic, and thus of process execution dynamics in the CLL system of Section 2. The proof dynamics of linear logic are derived from the computational content of the cut elimination proof, which defines a proof simplification strategy that removes (all) instances of the cut rule from a proof. However, the strategy induced by cut elimination is *non-deterministic* insofar as multiple simplification steps may apply to a given proof. Transposing this observation to CLL and other related systems, we observe that their operational semantics is does not prescribe a rigid evaluation order for processes. For instance, in the process cut $\{P \mid x \mid Q\}$, reduction is allowed in both P and Q. This is of course in line with reduction in process calculi (e.g., [49]). However, in logical-based systems this amounts to *don't care* non-determinism since, regardless of the evaluation order, confluence ensures that the same outcomes are produced (in opposition to *don't know* non-determinism which breaks confluence and is thus disallowed in purely logical systems). The design of the SAM arises from attempting to fix a purely sequential reduction strategy for CLL processes, such that only *one* process is allowed to execute at any given point in time, in the style of coroutines. To construct such a strategy, we forego the use of purely synchronous communication channels, which require a handshake between two concurrently executing processes, and so consider session channels as a kind of *buffered* communication medium (this idea has been explored in the context of linear logic interpretations of sessions in [25]), or queue, where one process can asynchronously write messages so that another may, subsequently, read. To ensure the correct directionality of communication, the queue has a write endpoint (on which a process may only write) and a read endpoint (along which only reads may be performed), such that at any given point in time a process can only hold one of two endpoints of a queue. Moreover, our design takes inspiration from insights related to polarisation and focusing in linear logic, grouping communication in sequences of positive (i.e. write) actions.

Allowing session channels to buffer message sequences, we may then model process execution by alternating between writer processes (that inject messages into the respective queues) and corresponding reader processes. Thus, the SAM

S	$::=$	(P, H)	State
H	$::=$	$\mathit{SessionRef} \to \mathit{SessionRec}$	Heap
R	$::=$	$x\langle q, P\rangle y$	Session Record
q	$::=$	$\mathsf{nil} \mid \mathit{Val}@q$	Queue
Val	$::=$	$\checkmark$	Close token
	$\mid$	$\#\mathsf{l}$	Choice label
	$\mid$	$\mathsf{clos}(x, P)$	Process Closure

Fig. 4: core SAM Components

must maintain a *heap* that tracks the queue contents of each session (and its endpoints), as well as the suspended processes. The construction of the core of the SAM is given in Figure 4. An execution state is simply a pair consisting of *the* running process P and the heap H. For technical reasons that are made clear in Sections 4 and 5, the process language used in the SAM differs superficially from that of CLL, but for the purposes of this overview we will use CLL process syntax. Later we show the two languages are equivalent in a strong sense.

A heap is a mapping between session identifiers and *session records* of the form $x\langle q, Q\rangle y$, denoting a session with write endpoint x and read endpoint y, with queue contents q and a suspended process Q, holding one of the two endpoints. If Q holds the read endpoint then it is suspended waiting for the process holding the write endpoint to fill the queue with data for it to read. If Q holds the write endpoint, then Q has been suspended *after* filling the queue and is now waiting for the reader process on y to empty the queue.

We adopt the convention of placing the write endpoint on the left and the read endpoint on the right. In general, session records in the SAM support a form of coroutines through their contained processes, which are called on and returned from multiple times over the course of the execution of the machine. A queue can either be empty (nil) or holding a sequence of values. A value is either a close session token ($\checkmark$), identifying the last output on a session; a choice label $\#\mathsf{l}$ or a process closure $\mathsf{clos}(x, P)$, used to model session send and receive. We overload the @ notation to also denote concatenation of queues.

Cut. We begin by considering how to execute a cut of the form $\mathsf{cut}\ \{P \mid x : A^{+} \mid Q\}$ where x is a positive type (in the sense of polarized logic [30]) in P. A positive type corresponds to a type denoting an *output* (or write) action, whereas a negative type denotes an *input* (or read) action. We maintain the invariant that in such a cut, P holds the write endpoint and Q the read endpoint. This means that the next action performed by P on the session will be to push some value onto the queue and, dually, the next action performed by Q on the session will be to read a value from the queue. In general, the holder of the write and read endpoint can change throughout execution.

Given the choice of either scheduling P or Q, we are effectively *forced* to schedule P *before* Q. Given that the cut introduces the (unique) session that is shared between the two processes, the only way for Q to exercise its read

capability on the session successfully is to wait for P to have exercised (at least some of) its write capability. If we were to schedule Q before P, the process might attempt to read a value from an empty queue, resulting in a stuck state of the SAM. Thus, the SAM execution rule for cut is:

$$(\mathsf{cut}\ \{P\ |x : A^{+}|\ Q\}, H) \Mapsto (P, H[x\langle \mathsf{nil}, \{y/x\}Q\rangle y]) \qquad [\text{SCut}]$$

The rule states that P is the process that is to be scheduled, adding the session record $x\langle \mathsf{nil}, Q\rangle y$ to the heap, which effectively suspends the execution of Q until P has exercised some of its write capabilities on the new session. Note that, in general, both P and Q can interact along many different sessions as both readers and writers before exercising any action on x (resp. y). However, they alone hold the freshly created endpoints x and y and so the next value sent along the session must come from P and Q is its intended receiver.

Channel Output. To execute an output of the form $\mathsf{send}\ x(z.R); Q$ in the SAM we simply lookup the session record for x and add to the queue a *process closure* containing R (which interacts along z), continuing with the execution of Q:

$$(\mathsf{send}\ x(z.R); Q, H[x\langle q, P\rangle y]) \Mapsto (Q, H[x\langle q@\mathsf{clos}(z, R), P\rangle y]) \qquad [\text{S}\otimes]$$

Note that the SAM *eagerly* continues to execute Q instead of switching to P, the holder of the read endpoint of the queue. This allows for the running process to perform all available writes before a context switch occurs.

Session Closure. Executing a close follows a similar spirit, but no continuation process exists and so execution switches to the process P holding the *read* endpoint y of the queue:

$$(\mathsf{close}\ x, H[x\langle q, P\rangle y]) \Mapsto (P, H[x\langle q@\checkmark, 0\rangle y]) \qquad [\text{S}\mathbf{1}]$$

The process P will eventually read the termination mark from the queue (triggering the deallocation of the session record from the heap):

$$(\mathsf{wait}\ y; P, H[x\langle \checkmark, 0\rangle y]) \Mapsto (P, H) \qquad [\text{S}\bot]$$

Note the requirement that $\checkmark$ be the final element of the queue.

Negative Action on Write Endpoint. As hinted above for the case of executing a cut, the SAM has a kind of *write bias* insofar as the process chosen to execute in a cut is that which holds the write endpoint for the newly created session. Since CLL processes use channels bidirectionally, the role of writer and reader on a channel (and thus the holder of the write and read endpoints of the queue) may be exchanged during execution. For instance, a process P may wish to send a value v to Q and then receive a response on the same channel. However, when considering a queue-based semantics, the execution of the input action *must not* obtain the value v, intended for Q. Care is therefore needed to ensure that v is received by the holder of the read endpoint of the queue *before* P is allowed to execute its input action (and so taking over the read endpoint). This notion is captured by the following rule, where $\mathcal{A}^{-}$ denotes any process

performing a negative polarity action (i.e., a wait, recv, case or, as we discuss later, a fwd $x\ y$ when x is a write endpoint with a negative polarity type):

$$(\mathcal{A}^-(x), H[x\langle q, Q\rangle y]) \Mapsto (Q, H[x\langle q, \mathcal{A}^-(x)\rangle y]) \qquad [\mathrm{S}-]$$

If the executing process is to perform a negative polarity action on a write endpoint x, the SAM context switches to Q, the holder of the read endpoint y of the session, and suspends the previously running process. This will now allow for Q to perform the appropriate inputs before execution of the action $\mathcal{A}^-$ resumes.

Channel Input. The rules for recv actions are as follows:

$$(\mathsf{recv}\ y(w{:}+); Q, H[x\langle \mathsf{clos}(z, R)@q, P\rangle y]) \Mapsto (Q, H[w\langle \mathsf{nil}, R\rangle z][x\langle q\rangle^s y]) \quad [\mathrm{S}\parr_+]$$
$$(\mathsf{recv}\ y(w{:}-); Q, H[x\langle \mathsf{clos}(z, R)@q, P\rangle y]) \Mapsto (R, H[z\langle \mathsf{nil}, Q\rangle w][x\langle q\rangle^s y]) \quad [\mathrm{S}\parr_-]$$

where $x\langle q\rangle^s y \triangleq$ if $(q = \mathsf{nil})$ then $y\langle q, P\rangle x$ else $x\langle q, P\rangle y$. The execution of an input action requires the corresponding queue to contain a process closure, denoting the process that interacts along the received channel w. In order to ensure that no inputs attempt to read from an empty queue, we must *branch* on the polarity of the communicated session (written $w{:}+$ and $w{:}-$ in the rules above): if the session has a *positive* type, then Q must take the *write* endpoint w of the newly generated queue (since Q uses the session with a dual type) and thus we execute Q and allocate a session record in the heap for the new session, with read endpoint z; if the exchanged session has a *negative* type, the converse holds and Q must take the *read* endpoint of the newly generated queue. In this scenario, we must execute R so that it may exercise its write capability on the queue and suspend Q in the new session record.

In either case, the session record for the original session is updated by removing the received message from the queue. Crucially, since processes are well-typed, if the resulting queue is empty then it must be the case that Q has no more reads to perform on the session, and so we *swap* the read and write endpoints of the session. This swap serves two purposes: first, it enables Q to perform writes if needed; secondly, and more subtly, it allows for the process, say, P, that holds the other endpoint of the queue to be resumed to perform its actions accordingly. To see how this is the case, consider that such a process will be suspended (due to rule [S−]) attempting to perform a negative action on the write endpoint of the queue. After the swap, the endpoint of the suspended process now matches its intended action. Since Q now holds the write endpoint, it will perform some number of positive actions on the session which end either in a close, which context switches to P, or until it attempts to perform a negative action on the write endpoint, triggering rule [S−] and so context switching to P.

Choice and Selection. The treatment of the additive constructs in the SAM is straightforward:

$$(\#\mathsf{l}\ x; Q, H[x\langle q, P\rangle y]) \Mapsto (Q, H[x\langle q@\#\mathsf{l}, P\rangle y]) \qquad [\mathrm{S}\oplus]$$
$$(\mathsf{case}\ y\ \{|\#\ell \in L{:}Q_\ell\}, H[x\langle \#\mathsf{l}@q, P\rangle y]) \Mapsto (Q_{\#\mathsf{l}}, H[x\langle q\rangle^s y]) \qquad [\mathrm{S}\&]$$

Sending a label $\#\mathsf{l}$ simply adds the $\#\mathsf{l}$ to the corresponding queue and proceeds with the execution, whereas executing a case reads a label from the queue and

continues execution of the appropriate branch. Since removing the label may empty the queue, we perform the same adjustment as in rules [S⅋+] and [S⅋−].

Forwarding. Finally, let us consider the execution of a forwarder (we overload the @ notation to also denote concatenation of queues):

$$(\mathsf{fwd}\ x^-\ y^+, H[z\langle q_1, Q\rangle x][y\langle q_2, P\rangle w]) \Mapsto (P, H[z\langle q_2 @ q_1, Q\rangle w]) \qquad \text{[Sfwd]}$$

A forwarder denotes the merging of two sessions x and y. Since the forwarder holds the read and write endpoints x and y, respectively, Q has written (through z) the contents of q_1, whereas the previous steps of the currently running process have written q_2. Thus, P is waiting to read $q_2 @ q_1$, justifying the rule above.

The reader may then wonder about other possible configurations of the SAM heap and how they interact with the forwarder. Specifically, what happens if y is of a positive type but a read endpoint of a queue, or, dually, if x is of a negative type but a write endpoint. The former case is *ruled out* by the SAM since the heap satisfies the invariant that any session record of the form $x{:}A\langle q, P\rangle y{:}A \in H$ is such that A must be of negative polarity or P is the inert process (which cannot be forwarded). The latter case is possible and is handled by rule [S−], since such a forward $\mathsf{fwd}\ x^-\ y^+$ stands for a process that wants to perform a *negative polarity* action on a *write* endpoint (or a positive action on a read endpoint).

3.1 On the Write-Bias of the SAM

Consider the following CLL process:

$$P \triangleq \mathsf{cut}\ \{P_1\ |a : \mathbf{1} \otimes \mathbf{1}|\ \{a/b\}Q_1\}$$

$$\begin{array}{ll} P_1 \triangleq \mathsf{send}\ a(y.P_2); P_3 & Q_1 \triangleq \mathsf{recv}\ b(x); Q_2 \\ P_2 \triangleq \mathsf{close}\ y & Q_2 \triangleq \mathsf{wait}\ x; Q_3 \\ P_3 \triangleq \mathsf{close}\ a & Q_3 \triangleq \mathsf{wait}\ b; 0 \end{array}$$

Let us walk through the execution trace of P:

(1) $(P, \emptyset) \Mapsto$ by [SCut]
(2) $(P_1, a\langle \mathsf{nil}, Q_1\rangle b) \Mapsto$ by [S⊗]
(3) $(P_3, a\langle \mathsf{clos}(y, P_2), Q_1\rangle b) \Mapsto$ by [S**1**]
(4) $(Q_1, a\langle \mathsf{clos}(y, P_2) @ \checkmark, 0\rangle b) \Mapsto$ by [S⅋−]
(5) $(P_2, y\langle \mathsf{nil}, Q_2\rangle x, a\langle \checkmark, 0\rangle b) \Mapsto$ by [S**1**]
(6) $(Q_2, y\langle \checkmark, 0\rangle x, a\langle \checkmark, 0\rangle b) \Mapsto$ by [S⊥]
(7) $(Q_3, a\langle \checkmark, 0\rangle b) \Mapsto$ by [S⊥]
(8) $(0, \emptyset)$

The SAM begins in the state on line (1) above, executing the cut. Since the type of a is positive, we execute P_1, and allocate the session record, suspending Q_1, resulting in the state on line (2). Since P_1 is a write action on a write endpoint, we proceed via the [S⊗] rule, resulting in the SAM configuration in line (3), executing P_3 and adding a closure containing P_2 to the session queue with write endpoint a. Executing P_3 (3), a close action, requires adding the ✓

to the queue and context switching to the process Q_1, now ready to receive the sent value. The applicable rule is now (4) [S⅋_], and so execution will context switch to P_2 after creating the session record for the new session with endpoints y and x. P_2 will execute and the machine ends up in state (6) followed by (7), which consume the appropriate ✓ and deallocate the session records.

Note how after executing the send action of P_1 we eagerly execute the positive action in P_3 rather than context switching to Q_1. While in this particular process it would have been safe to execute the negative action in Q_1, switch to P_2 and then back to Q_2, we would now need to somehow context switch to P_3 *before* continuing with the execution of Q_3, or execution would be stuck. However, the relationship between P_3 and Q_2 is unclear at best. Moreover, if the continuation of Q_1 were of the form $\mathsf{wait}\ b; \mathsf{wait}\ x; 0$, the context switch after the execution of P_2 would have to execute P_3, or the machine would also be in a stuck state.

3.2 Illustrating Forwarding

To better illustrate the way in which $\mathsf{fwd}\ x^-\ y^+$ effectively stands for a negative action, consider the following CLL process (to simplify the execution trace we assume the existence of output and input of integers typed as $\mathsf{int} \otimes A$ and $\overline{\mathsf{int}} \parr A$, respectively, eliding the need for process closures in this example):

$$P \triangleq \mathsf{cut}\ \{P_1\ |b : \overline{\mathsf{int}} \parr \overline{\mathsf{int}} \parr \mathbf{1}|\ \{b/c\}\mathsf{cut}\ \{Q_1\ |a : \mathsf{int} \otimes \overline{\mathsf{int}} \parr \mathbf{1}|\ \{a/d\}R_1\}\}$$

$$\begin{array}{lll} P_1 \triangleq \mathsf{recv}\ b(x); P_2 & Q_1 \triangleq \mathsf{send}\ a(1); Q_2 & R_1 \triangleq \mathsf{recv}\ d(y); R_2 \\ P_2 \triangleq \mathsf{recv}\ b(z); P_3 & Q_2 \triangleq \mathsf{send}\ c(3); Q_3 & R_2 \triangleq \mathsf{send}\ d(2); R_3 \\ P_3 \triangleq \mathsf{close}\ b & Q_3 \triangleq \mathsf{fwd}\ a\ c & R_3 \triangleq \mathsf{wait}\ d; 0 \end{array}$$

If we consider the execution of P we observe:

(1) $(P, \emptyset) \mapsto$	by [SCut]
(2) $(\mathsf{cut}\ \{Q_1\ \|a\|\ \{a/d\}R_1\}, c\langle \mathsf{nil}, P_1\rangle b) \mapsto$	by [SCut]
(3) $(Q_1, a\langle \mathsf{nil}, R_1\rangle d, c\langle \mathsf{nil}, P_1\rangle b) \mapsto$	by [S⊗]
(4) $(Q_2, a\langle 1, R_1\rangle d, c\langle \mathsf{nil}, P_1\rangle b) \mapsto$	by [S⊗]
(5) $(\mathsf{fwd}\ a\ c, a\langle 1, R_1\rangle d, c\langle 3, P_1\rangle b) \mapsto$	by [S−]
(6) $(R_1, a\langle 1, Q_3\rangle d, c\langle 3, P_1\rangle b) \mapsto$	by [S⅋]
(7) $(R_2, d\langle \mathsf{nil}, Q_3\rangle a, c\langle 3, P_1\rangle b) \mapsto$	by [S⊗]
(8) $(R_3, d\langle 2, Q_3\rangle a, c\langle 3, P_1\rangle b) \mapsto$	by [S−]
(9) $(\mathsf{fwd}\ a\ c, d\langle 2, R_3\rangle a, c\langle 3, P_1\rangle b) \mapsto$	by [Sfwd]
(10) $(P_1, d\langle 3@2, R_3\rangle b) \mapsto$	by [S⅋]
(11) $(P_2, b\langle 2, R_3\rangle d) \mapsto$	by [S⅋]
(12) $(P_3, b\langle \mathsf{nil}, R_3\rangle d) \mapsto$	by [S**1**]
(13) $(R_3, b\langle ✓, R_3\rangle d) \mapsto$	by [S⊥]
(14) $(0, \emptyset)$	

The first four steps of the execution of P allocate the two session records and the writes by Q_1 and Q_2 takes place. We are now in configuration (5), where $Q_3 = \mathsf{fwd}\ a^-\ c^+$ is to execute and a is a write endpoint of a queue assigned a negative type ($\overline{\mathsf{int}} \parr \mathbf{1}$). This forwarder stands for a process performing a negative action on a write endpoint (i.e., P_1) and so context switching is required, rule [S−] applies and the SAM context switches to R_1, suspending Q_3 until the

forward can be performed. After R_1 receives (6) and the queue endpoints a and d are swapped (7), R_2 executes and then rule [S−] applies (8), context switching back to Q_3. Since the queue endpoints are now flipped, rule [Sfwd] now applies (9), collapsing the two session records (via queue concatenation) and proceeding with the execution of P_1, P_2, P_3 and R_3 (10-14). Note the correct ordering in which the sent values are dequeued, where 3 is read before 2, as intended.

Discussion. The core execution rules for the SAM are summarized in Figure 5. At this point, the reader may wonder just how reasonable the SAM's evaluation strategy is. Our evaluation strategy is devised to be a deterministic, sequential strategy, where exactly one process is executing at any given point in time, supported by a queue-based buffer structure for channels and a heap for session records. Moreover, taking inspiration from focusing and polarized logic, we adopt a write-biased stance and prioritize (bundles of) write actions over reads, where suspended processes hold the read endpoint of queues while waiting for the writer process to fill the queue, and hold write endpoints of queues *after* filling them, waiting for the reader process to empty the queue.

While this latter point seems like a reasonable way to ensure that inputs never get stuck, it is not immediately obvious that the strategy is sound wrt the more standard (asynchronous) semantics of CLL and related languages, given that processes are free to act on multiple sessions. Thus, the write-bias of the cut rule (and the overall SAM) does not necessarily mean that the process that is chosen to execute will immediately perform a write action on the freshly cut session x. In general, such a process may perform multiple write or read actions on many other sessions before performing the write on x, meaning that multiple context switches may occur. Given this, it is not obvious that this strategy is adequate insofar as preserving the correctness properties of CLL in terms of soundness, progress and type preservation. The remainder of this paper is devoted to establishing this correspondence in a precise technical sense.

4 CLLB: A Buffered Formulation of CLL

There is a substantial gap between the language CLL, presented in an abstract algebraic style, and its operational semantics, defined by equational and rewriting systems, and an abstract machine as the SAM, a deterministic state machine manipulating several low level structures. Therefore, even if the core SAM structure and transition rules are fairly simple, proving its correctness is more challenging and technically involved, and require progressive build up. Therefore, we first bridge between CLL and SAM via a intermediate logical language CLLB, which extends CLL with a buffered cut construct.

$$\mathsf{cut}\ \{P \mid a : A\ [q]\ b : B \mid Q\}$$

The buffered cut construct models interaction via a "message queue" with two polarised endpoints a and b, held respectively by the processes P and Q. A polarised endpoint has the form x or $\overline{x}$. The endpoint marked $\overline{x}$ is the only allowing writes, the unmarked y is the only one allowing reads, exactly one of

$$(\mathsf{cut}\ \{P\ |x : A^+|\ Q\}, H) \Mapsto (P, H[x\langle \mathsf{nil}, \{y/x\}Q\rangle y]) \quad [\text{SCut}]$$

$$(\mathsf{fwd}\ x\ y, H[z\langle q_1, Q\rangle x][y\langle q_2, P\rangle w]) \Mapsto (P, H[z\langle q_2 @ q_1, Q\rangle w]) \quad [\text{Sfwd}]$$

$$(\mathsf{close}\ x, H[x\langle q, P\rangle y]) \Mapsto (P, H[x\langle q @ \checkmark, 0\rangle y]) \quad [\text{S}\mathbf{1}]$$

$$(\mathsf{wait}\ y; P, H[x\langle \checkmark, 0\rangle y]) \Mapsto (P, H) \quad [\text{S}\bot]$$

$$(\mathcal{A}^-(x), H[x\langle q, Q\rangle y]) \Mapsto (Q, H[x\langle q, \mathcal{A}^-(x)\rangle y]) \quad [\text{S}-]$$

$$(\mathsf{send}\ x(z.R); Q, H[x\langle q, P\rangle y]) \Mapsto (Q, H[x\langle q @ \mathsf{clos}(z, R), P\rangle y]) \quad [\text{S}\otimes]$$

$$(\mathsf{recv}\ y(w : +); Q, H[x\langle \mathsf{clos}(z, R) @ q, P\rangle y]) \Mapsto (Q, H[w\langle \mathsf{nil}, R\rangle z][x\langle q\rangle^s y]) \quad [\text{S}\parr]$$

$$(\mathsf{recv}\ y(w : -); Q, H[x\langle \mathsf{clos}(z, R) @ q, P\rangle y]) \Mapsto (R, H[z\langle \mathsf{nil}, Q\rangle w][x\langle q\rangle^s y]) \quad [\text{S}\parr]$$

$$(\#\mathsf{l}\ x; Q, H[x\langle q, P\rangle y]) \Mapsto (Q, H[x\langle q @ \#\mathsf{l}, P\rangle y]) \quad [\text{S}\oplus]$$

$$(\mathsf{case}\ y\ \{|\#\ell \in L{:}Q_\ell\}, H[x\langle \#\mathsf{l} @ q, P\rangle y]) \Mapsto (Q_{\#\mathsf{l}}, H[x\langle q\rangle^s y]) \quad [\text{S}\&]$$

$$N.B. : x\langle q\rangle^s y \triangleq \text{if } (q = \mathsf{nil}) \text{ then } y\langle q, P\rangle x \text{ else } x\langle q, P\rangle y$$

Fig. 5: The core SAM Transition Rules

the two endpoints is marked. The endpoints types A, B are of course related but do not need to be exact duals, the type of the writer endpoint may be advanced in time wrt the type of the reader endpoint, reflecting the messages already enqueued but not yet consumed. If the queue is empty, we have $A = \overline{B}$. Thus a buffered cut with empty queue corresponds to the basic cut of CLL.

$$\mathsf{cut}\ \{P\ |x : A|\ Q\} \equiv \mathsf{cut}\ \{P\ |\overline{x} : A\ [\mathsf{nil}]\ y : \overline{A}|\ \{y/x\}Q\} \quad (A+)$$

The queue q stores values V defined by

$$\begin{array}{rcl} V ::= \checkmark & \text{(Close token)} & \mid\ \#\mathsf{l}\ \text{(Selection Label)} \\ \mid\ \mathsf{clos}(x, P) & \text{(Linear Closure)} & \mid\ \mathsf{clos!}(x, P)\ \text{(Exponential Closure)} \end{array}$$

$$q ::= \mathsf{nil} \mid V \mid V @ q \quad \text{(Queue)}$$

We use @ to also denote (associative) concatenation operation of queues, with unit nil. Enqueue and dequeue operations occur respectively on the lhs and rhs.

The type system CLLB is obtained from CLL by replacing [TCut] with the typing rules (and symmetric ones) in Fig. 6. We distinguish the type judgements as $P \vdash \Delta; \Gamma$ for CLL and $P \vdash^{\mathsf{B}} \Delta; \Gamma$ for CLLB. The [TCutB] rule sets the endpoints mode based in the cut type polarity, applicable whenever the queue is empty. The remaining rules relate queue contents with their corresponding (positive action) processes. For instance, rule [Tcut-$\otimes$] can be read bottom-up as stating that typing processes mediated by a queue containing a process closure $\mathsf{clos}(y, R)$ amounts to typing the process that will emit the session y (bound to R), interacting with the queue with the closure removed. Rules [Tcut-$\oplus$] and [Tcut!] apply a similar principle to the other possible queue contents. In [Tcut-$\mathbf{1}$] and [Tcut!] the write endpoint is typed $\emptyset$, as the sender has terminated (0).

$$\frac{P \vdash^{\mathsf{B}} \Delta', x : \overline{A}; \Gamma \quad Q \vdash^{\mathsf{B}} \Delta, y : A; \Gamma}{\mathsf{cut}\ \{P\ |x : \overline{A}\ [\mathsf{nil}]\ \overline{y} : A|\ Q\}\ \vdash^{\mathsf{B}} \Delta', \Delta; \Gamma}\ (\text{A positive}) \quad [\text{TcutB}]$$

$$\frac{\mathsf{cut}\ \{\mathsf{close}\ x\ |\overline{x} : \mathbf{1}\ [q]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}{\mathsf{cut}\ \{0\ |\overline{x} : \emptyset[q@\checkmark]y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}\ [\text{Tcut-}\mathbf{1}]$$

$$\frac{\mathsf{cut}\ \{\mathsf{send}\ x(y.R); P\ |\overline{x} : T \otimes A\ [q]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}{\mathsf{cut}\ \{P\ |\overline{x} : A\ [q@\mathsf{clos}(y, R)]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}\ [\text{Tcut-}\otimes]$$

$$\frac{\mathsf{cut}\ \{\#\mathsf{l}\ x; P\ |\overline{x} : \oplus_{\ell \in L} A_\ell\ [q]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}{\mathsf{cut}\ \{P\ |\overline{x} : A_{\#\mathsf{l}}\ [q@\#\mathsf{l}]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}\ [\text{Tcut-}\oplus]$$

$$\frac{\mathsf{cut}\ \{!x(z); P\ |\overline{x} : !A\ [q]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}{\mathsf{cut}\ \{0\ |\overline{x} : \emptyset\ [q@\mathsf{clos!}(z, P)]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}\ [\text{Tcut!}]$$

Fig. 6: Additional typing rules for CLLB.

$$\mathsf{cut}\ \{Q\ |a : A[q]b : B|\ P\}\ \equiv^{\mathsf{B}} \mathsf{cut}\ \{Q\ |b : B[q]a : A|\ P\} \qquad [\text{comm}]$$

$$\mathsf{cut}\ \{P\ |x[q]y|\ (Q\ ||\ R)\} \equiv^{\mathsf{B}} (\mathsf{cut}\ \{P\ |x[q]y|\ Q\})\ ||\ R \qquad [\text{CM}]$$

$$P\ |x[q]z|\ (\mathsf{cut}\ \{Q\ |y[p]w|\ R\}) \equiv^{\mathsf{B}} \mathsf{cut}\ \{(\mathsf{cut}\ \{P\ |x[q]z|\ Q\})\ |y[p]w|\ R\} \qquad [\text{CC}]$$

$$\mathsf{cut}\ \{P\ |z[q]w|\ (\mathsf{cut!}\ \{y.Q\ |!x|\ R\})\} \equiv^{\mathsf{B}} \mathsf{cut!}\ \{y.Q\ |!x|\ (\mathsf{cut}\ \{P\ |z[q]w|\ R\})\} \qquad [\text{CC!}]$$

$$\mathsf{cut!}\ \{y.P\ |!x|\ (\mathsf{cut}\ \{Q\ |z[q]w|\ R\})\} \equiv^{\mathsf{B}} \mathsf{cut}\ \{(y.P\ |!x|\ Q)\ |z[q]w|\ (\mathsf{cut!}\ \{y.P\ |!x|\ R\})\} \qquad [\text{D-C!}]$$

Fig. 7: Additional structural congruence rules for CLLB.

Structural congruence for B (noted $\equiv^{\mathsf{B}}$) is obtained by extending $\equiv$ with commutative conversions for the buffered cut, listed in Fig. 7. The following provisos apply: [CM] $y \in \mathsf{fn}(Q)$; in [CC] $y, z \in \mathsf{fn}(Q)$; in [CC!] $x \notin \mathsf{fn}(P)$. Accordingly, reduction for B (noted $\rightarrow^{\mathsf{B}}$) is obtained by replacing the $\rightarrow$ rules [fwd], [$\mathbf{1}\bot$], [$\otimes\parr$] and [$\oplus\&$] by the rules in Fig. 8. Essentially each principal cut reduction rule of CLL is replaced by a pair of "positive" ($\rightarrow_p$) / "negative" ($\rightarrow_n$) reduction rules that allow processes to interact asynchronously via the queue, that is, positive process actions (corresponding to positive types) are non-blocking. For example, the rule [$\otimes$] for send appends a session closure to the tail of the queue (rhs) and the rule for receive pops a session closure from the head of the queue (lhs). Notice that positive rules are enabled only if the relevant endpoint is in write mode ($\overline{x}$), and negative rules are enabled only if the relevant endpoint is in read mode (y). In [$\parr$] above the target cuts endpoint polarities depends on the types of the composed processes. To uniformly express the appropriate marking of endpoint polarities we define some convenient abbreviations:

$$\mathsf{cut}\ \{Q\ |\overline{z}\ [q_1]\ x|\ \mathsf{fwd}\ x\ y\ |\overline{y}\ [q_2]\ w|\ P\} \to^{\mathsf{B}} \mathsf{cut}\ \{Q\ |\overline{z}\ [q_2@q_1]\ w|\ P\} \quad [\text{fwdp}]$$

$$\mathsf{cut}\ \{\mathsf{close}\ x\ |\overline{x}\ [q]\ y|\ Q\} \to^{\mathsf{B}} \mathsf{cut}\ \{0\ |\overline{x}\ [q@\checkmark]\ y|\ Q\} \quad [\mathbf{1}]$$

$$\mathsf{cut}\ \{0\ |\overline{x}\ [\checkmark]\ y|\ \mathsf{wait}\ y; P\} \to^{\mathsf{B}} P \quad [\bot]$$

$$\mathsf{cut}\ \{\mathsf{send}\ x(z.P); Q\ |\overline{x}\ [q]\ y|\ R\} \to^{\mathsf{B}} \mathsf{cut}\ \{Q\ |\overline{x}\ [q@\mathsf{clos}(z,P)]|\ y|\ R\} \quad [\otimes]$$

$$\mathsf{cut}\ \{Q\ |\overline{x}\ [\mathsf{clos}(z,P)@q]\ y|\ \mathsf{recv}\ y(w); R\} \to^{\mathsf{B}} \mathsf{cut}\ \{Q\ |\overline{x}\ [q]\ y|\ \mathsf{cut}\ \{P\ |z\ [\mathsf{nil}]\ w|\ R\}^{\mathsf{p}}\}^{\mathsf{p}} \quad [\parr]$$

$$\mathsf{cut}\ \{\#\mathsf{l}\ x; P\ |\overline{x}\ [q]\ y|\ R\} \to^{\mathsf{B}} \mathsf{cut}\ \{Q\ |\overline{x}\ [q@\#\mathsf{l}]|\ y|\ R\} \quad [\oplus]$$

$$\mathsf{cut}\ \{Q\ |\overline{x}\ [\mathsf{l}@q]\ y|\ \mathsf{case}\ y\ \{|\#\ell \in L{:}P_\ell\}\ \} \to^{\mathsf{B}} \mathsf{cut}\ \{Q\ |x\ [q]\ y|\ P_{\#\mathsf{l}}\}^{\mathsf{p}} \quad [\&]$$

$$\mathsf{cut}\ \{!x(z); P\ |\overline{x}\ [q]\ y|\ Q\} \to^{\mathsf{B}} \mathsf{cut}\ \{0\ |\overline{x}\ [q@\mathsf{clos!}(z,P)]|\ y|\ Q\} \quad [!]$$

$$\mathsf{cut}\ \{0\ |\overline{x}\ [\mathsf{clos!}(y,P)]\ y|\ ?y; Q\} \to^{\mathsf{B}} \mathsf{cut!}\ \{y.P\ |!x|\ Q\} \quad [?]$$

Fig. 8: Reduction $P \to^{\mathsf{B}} Q$.

Definition 4.1 (Setting polarities).

$$\mathsf{cut}\ \{Q\ |a : A[\mathsf{nil}]b : B|\ P\}^{\mathsf{p}} \triangleq \mathit{if}\ {+}A\ \mathit{then}\ \mathsf{cut}\ \{Q\ |\overline{a} : A[\mathsf{nil}]b : B|\ P\}\ \mathit{else}\ \mathsf{cut}\ \{Q\ |a : A[\mathsf{nil}]\overline{b} : B|\ P\}$$

$$\mathsf{cut}\ \{Q\ |\overline{a} : A[q]b : B|\ P\}^{\mathsf{p}} \triangleq \mathsf{cut}\ \{Q\ |\overline{a} : A[q]b : B|\ P\} \quad (q \neq \mathsf{nil})$$

The following definition then formalizes the intuition given above about how to encode processes of CLL into processes of CLLB.

Definition 4.2 (Embedding). *Let $P \vdash \Delta; \Gamma$. $P^\dagger$ is the* B *process such that*

$$(\mathsf{cut}\ \{P\ |x : A|\ Q\})^\dagger \triangleq \mathsf{cut}\ \{P^\dagger\ |x : A\ [\mathsf{nil}]\ y : \overline{A}|\ (\{y/x\}Q)^\dagger\}^{\mathsf{p}}$$

homomorphically defined in the remaining constructs. Clearly $P^\dagger \vdash^{\mathsf{B}} \Delta; \Gamma$.

4.1 Preservation and Progress for CLLB

In this section, we prove basic safety properties of CLLB: Preservation (Theorem 4.1) and Progress (Theorem 4.2). To reason about type derivations involving buffered cuts, we formulate some auxiliary inversion principles that allow us, by aggregating sequences of application of [TCut-$*$] rules of CLLB, to talk in a uniform way about typing of values in queues and typing of processes connected by queues. To assert typing of queue values c we use judgments the form $\Gamma; \Delta \vdash c{:}E$, where E is a either a type or a one hole type context, defined by

$$E ::= \Box \mid T \mid T \parr E \mid \ \&_{\ell \in L}\ E_\ell$$

where in $\&_{\ell \in L} E_\ell$ only branch type $E_{\#\mathsf{l}}$ for some selected label $\#\mathsf{l} \in L$ is a one hole context (to plug the continuation type); only the branch chosen by the

selected label in a queue is relevant to type next queue values. We identify the selected branch in the type by tagging it with the corresponding label $\#\mathsf{l}$ thus $\&_{\ell \in L} E_\ell[\#\mathsf{l}]$. We then introduce the following typing rules for queue values.

Definition 4.3 (Typing of Queue Values).

$$\frac{}{\Gamma; \vdash_{\mathtt{val}} \checkmark : \bot} \qquad \frac{P \vdash^{\mathsf{B}} \Delta, z : T; \Gamma}{\Gamma; \Delta \vdash_{\mathtt{val}} \mathsf{clos}(z, P) : \overline{T} \parr E}$$

$$\frac{\&_{\ell \in L} E_\ell[\#\mathsf{l}]}{\Gamma; \Delta \vdash_{\mathtt{val}} \#\mathsf{l} : \&_{\ell \in L} E_\ell} \qquad \frac{P \vdash^{\mathsf{B}} z : A; \Gamma}{\Gamma; \vdash_{\mathtt{val}} \mathsf{clos!}(z, P) : ?\overline{A}}$$

Given a sequence of k one hole queue value types E_i and a type A, we denote by $\mathbf{E}_k; A$ the type $E_1[E_2[...E_k[A]]]$. Queue value types allow us to talk in a uniform way about the type a receiver processes compatible with the types of enqueued values, as characterized by the following Lemma 4.1 and Lemma 4.2.

Lemma 4.1 (Non-full). *For $P \neq 0$ the rule below is admissible and invertible:*

$$\frac{P \vdash^{\mathsf{B}} \Delta_P, x{:}A; \Gamma \quad Q \vdash^{\mathsf{B}} \Delta_Q, y{:}B; \Gamma \quad q = \overline{c_k} \quad B = \mathbf{E}_k; \overline{A} \quad \Gamma; \Delta_i \vdash c_i{:}E_i \quad -B}{\mathsf{cut}\ \{P\ |\overline{x}{:}A\ [q]\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_1, ..., \Delta_k; \Gamma}$$

Notice that a session type, as defined by a CLL proposition, may terminate in either $\mathbf{1}$, $\bot$ or an exponential type $!A/?A$. We then also have

Lemma 4.2 (Full). *The proof rules below are admissible:*

$$\frac{Q \vdash^{\mathsf{B}} \Delta_Q, y : B; \Gamma \quad \Gamma; \Delta_i \vdash c_i : E_i \quad B = \mathbf{E}_k; \bot \quad c_k = \checkmark \quad -B}{\mathsf{cut}\ \{0\ |\overline{x} : \emptyset\ [\overline{c_k}]\ y : B|\ Q\} \vdash^{\mathsf{B}} \Delta_Q, \Delta_1, \ldots, \Delta_k; \Gamma}$$

$$\frac{Q \vdash^{\mathsf{B}} \Delta_Q, y{:}B; \Gamma \quad \Gamma; \Delta_i \vdash c_i{:}E_i \quad B = \mathbf{E}_{k-1}; C \quad \Gamma \vdash c_k = \mathsf{clos!}(z, R){:}C \quad -B}{\mathsf{cut}\ \{0\ |\overline{x} : \emptyset\ [\overline{c_k}]\ y : B|\ Q\} \vdash^{\mathsf{B}} \Delta_Q, \Delta_1, \ldots, \Delta_k, \Gamma}$$

Moreover, one of them must apply for inverting the judgment in the conclusion.

Theorem 4.1 (Preservation). *Let $P \vdash^{\mathsf{B}} \Delta; \Gamma$.*
(1) If $P \equiv^{\mathsf{B}} Q$, then $Q \vdash^{\mathsf{B}} \Delta; \Gamma$. (2) If $P \rightarrow^{\mathsf{B}} Q$, then $Q \vdash^{\mathsf{B}} \Delta; \Gamma$.

A process P is *live* if and only if $P = \mathcal{C}[Q]$, for some static context $\mathcal{C}$ (the hole lies within the scope of static constructs mix, cut) and Q is an action process. We first show that a live process either reduces or offers an interaction on a free name. The observability predicate defined in Fig. 9 (cf. [63]) characterises interactions of a process with the environment.

Lemma 4.3 (Liveness). *Let $P \vdash^{\mathsf{B}} \Delta; \Gamma$ be live. Either $P \downarrow_x$ or $P \rightarrow^{\mathsf{B}}$.*

Theorem 4.2 (Progress). *Let $P \vdash \emptyset; \emptyset$ be a live process. Then, $P \rightarrow^{\mathsf{B}}$.*

$$\frac{}{\mathsf{fwd}\ x\ y\downarrow_x}\ [\text{fwd}] \quad \frac{s(\mathcal{A})=x}{\mathcal{A}\downarrow_x}\ [\mathcal{A}] \quad \frac{P\equiv Q \quad Q\downarrow_x}{P\downarrow_x}\ [\equiv] \quad \frac{P\downarrow_x}{(P\ ||\ Q)\downarrow_x}\ [\text{mix}]$$

$$\frac{P\downarrow_x \quad x\neq y}{(P\ |y[q]x|\ Q)\downarrow_x}\ [\text{cut}] \quad \frac{Q\downarrow_x \quad x\neq y}{(z.P\ |!y|\ Q)\downarrow_x}\ [\text{cut!}]$$

Fig. 9: Observability Predicate $P\downarrow_x$.

4.2 Correspondence between CLL and CLLB

In this section we establish the correspondence between reduction in CLL and CLLB, proving that the two languages simulate each other in a tight sense. Intuitively, the correspondence shows that CLLB allows some positive actions to be buffered ahead of reception, while in CLL a single positive action synchronises with the corresponding dual in one step, or a forward reduction takes place.

We write a reduction $P\rightarrow^{\mathsf{B}}Q$ as $P\rightarrow^{\mathsf{B}p}Q$ if the reduced action is positive, $P\rightarrow^{\mathsf{B}n}Q$ if the reduced action is negative (we consider [call] negative), $P\rightarrow^{\mathsf{B}a}Q$ if the reduced action is a forwarder, and $P\rightarrow^{\mathsf{Bap}}Q$ if the reduced action is positive or a forwarder. We also write $P\rightarrow^{\mathsf{Br}}Q$ for positive action followed by a matching negative action on the same cut with an initially empty queue.

Lemma 4.4. *The following commutations of reductions hold.*

1. *Let* $P_1\rightarrow^{\mathsf{B}p}S\rightarrow^{\mathsf{B}n}P_2$. *Either* $P_1\rightarrow^{\mathsf{Br}}P_2$, *or* $P_1\rightarrow^{\mathsf{B}n}S'\rightarrow^{\mathsf{B}p}P_2$ *for some* S'.
2. *Let* $P_1\rightarrow^{\mathsf{B}a}S\rightarrow^{\mathsf{B}n}P_2$. *Then* $P_1\rightarrow^{\mathsf{B}n}S'\rightarrow^{\mathsf{B}a}P_2$ *for some* S'.
3. *If* $P_1\rightarrow^{\mathsf{Bap}}S\rightarrow^{\mathsf{B}n}P_2$, *either* $P_1\rightarrow^{\mathsf{Br}}P_2$, *or* $P_1\rightarrow^{\mathsf{B}n}S'\rightarrow^{\mathsf{Bap}}P_2$ *for some* S'.
4. *Let* $P_1\rightarrow^{\mathsf{Bap}}N\overset{\epsilon}{\Rightarrow}{}^{\mathsf{B}a}S\rightarrow^{\mathsf{Br}}P_2$. *Either* $P_1\overset{\epsilon}{\rightarrow}{}^{\mathsf{B}a}N$ *or* $P_1\rightarrow^{\mathsf{Br}}S'\rightarrow^{\mathsf{Bap}}P_2$ *for some* S'.

Lemma 4.5 (Simulation). *Let* $P\vdash\emptyset;\emptyset$. *If* $P\rightarrow Q$ *then* $P^\dagger\Rightarrow^{\mathsf{B}}Q^\dagger$.

Proof. Each cut reduction of CLL is either simulated by two reduction steps of B in sequence or by a [fwd] reduction.

The following lemma identifies that in CLLB, a sequence of positive actions (or forwards) followed by a negative action can always be commuted either by pulling out the negative action first, followed by the sequence of positive actions and forwards; having the negative action follow a positive action on the same channel and then performing the remaining actions; or by first performing a sequence of forward actions, the output and input on the relevant session and then the remaining actions.

Lemma 4.6 (Simulation). *Let* $P\vdash^{\mathsf{B}}\emptyset;\emptyset$. *If* $P\Rightarrow^{\mathsf{Bap}}\rightarrow^{\mathsf{B}n}Q$ *then (1)* $P\rightarrow^{\mathsf{B}n}R$ *and* $R\Rightarrow^{\mathsf{Bap}}Q$ *for some* R, *or; (2)* $P\rightarrow^{\mathsf{Br}}R$ *and* $R\Rightarrow^{\mathsf{Bap}}Q$ *for some* R, *or; (3)* $P\overset{\epsilon}{\Rightarrow}{}^{\mathsf{B}a}\rightarrow^{\mathsf{Br}}R$ *and* $R\Rightarrow^{\mathsf{Bap}}Q$ *for some* R.

Theorem 4.3 (Operational correspondence CLL-CLLB). *Let* $P\vdash\emptyset;\emptyset$.

1. *If* $P\Rightarrow R$ *then* $P^\dagger\Rightarrow^{\mathsf{B}}R^\dagger$.

2. If $P^\dagger(\Rightarrow^{\mathsf{Bap}} \rightarrow^{\mathsf{Bn}})^* Q$ *then there is* R *such that* $P \Rightarrow R$ *and* $R^\dagger \Rightarrow^{\mathsf{Bap}} Q$.

Due to the progress property for CLLB (Theorem 4.2) and because queues are bounded by the size of positive/negative sections in types, after a sequence of positive or forwarder reductions a negative reduction consuming a queue value must occur. Theorem 4.3(2) states that every reduction sequence in CLLB is simulated by a reduction sequence in CLL up to some anticipated forwarding and buffering of positive actions. Our results imply that every reduction path in CLLB maps to a reduction path in CLL in which every negative reduction step in the former is mapped, in order, to a cut reduction step in the latter.

5 Correctness of the core SAM

We now prove that every execution trace of the core SAM defined in Fig. 5 represents a correct process reduction sequence CLLB (and therefore of CLL, in the light of Theorem 4.3), first for the language without exponentials and mix, which will be treated in Section 6. In what follows, we annotate endpoints of session records with their types (e.g. as $x{:}A\langle q, P\rangle y{:}B$), these annotations are not needed to guide the operation of the SAM, but convenient for the proofs; they will be omitted when not relevant or are obvious from the context. We first define a simple encoding of well-typed CLLB processes to SAM states.

Definition 5.1 (Encode). *Given* $P \vdash^{\mathsf{B}} \emptyset$ *we define* $enc(P) = \mathcal{C}$ *as* $enc(P, \emptyset) \overset{\mathsf{cut}^*}{\mapsto} \mathcal{C}$ *where* $enc(P, H) \overset{\mathsf{cut}^*}{\mapsto} \mathcal{C}$ *is defined by the rules*

$$\frac{enc(P(x), H[x{:}A\langle q, Q\rangle y{:}B]) \overset{\mathsf{cut}^*}{\mapsto} \mathcal{C}}{enc(\mathsf{cut}\ \{P\ |\overline{x}{:}A[q]\ y{:}B]|\ Q\}, H) \overset{\mathsf{cut}^*}{\mapsto} \mathcal{C}} \quad (A+)$$

$$\frac{enc(Q(y), H[x{:}A\langle q, P\rangle y{:}B]) \overset{\mathsf{cut}^*}{\mapsto} \mathcal{C}}{enc(\mathsf{cut}\ \{P\ |\overline{x}{:}A\ [q]\ y{:}B]|\ Q\}, H) \overset{\mathsf{cut}^*}{\mapsto} \mathcal{C}} \quad (A- \text{ or } P = 0)$$

$$enc(\mathcal{A}, H) \overset{\mathsf{cut}^*}{\mapsto} (\mathcal{A}, H) \qquad (A \in \mathcal{A})$$

Notice that $enc(P)$ maximally applies the SAM execution rule for cut to $(P, \emptyset)$ until an action is reached. Clearly, for any $P \vdash^{\mathsf{B}} \emptyset$, if $enc(P) = \mathcal{C}$ then $P \mapsto^* \mathcal{C}$. Also, if all cuts in a state C have empty queues then there is a process Q of CLL such that $enc(Q^\dagger) = C$. We then have

Theorem 5.1 (Soundness wrt CLLB). *Let* $P \vdash^{\mathsf{B}} \emptyset$.
If $enc(P) \mapsto D \overset{\mathsf{cut}^*}{\mapsto} \mathcal{C}$ *then there is* Q *such that* $P \rightarrow \cup \equiv Q$ *and* $\mathcal{C} = enc(Q)$.

We can then combine soundness with the operational correspondence between CLL and CLLB (Theorem 4.3) to obtain an overall soundness result for the SAM with respect to CLL:

Theorem 5.2 (Soundness wrt CLL). *Let* $P \vdash^{\mathsf{B}} \emptyset$.
1. If $enc(P) \overset{*}{\mapsto} \overset{\mathsf{cut}^*}{\mapsto} C$ *there is* Q *such that* $P \Rightarrow \cup \equiv Q$ *and* $\mathcal{C} = enc(Q)$.
2. Let $P \vdash \emptyset$. *If* $enc(P^\dagger) \overset{*}{\mapsto} enc(Q^\dagger)$ *then* $P \Rightarrow Q$.

In Definition 5.2 we identify readiness, the fundamental invariant property of SAM states, key to prove progress of its execution strategy. Readiness means that any running process holding an endpoint of negative type, and thus attempting to execute a negative action (e.g., a receive or offer action) on it, will always find an appropriate value (resp. a closure or a label) to be read in the appropriate session queue. No busy waiting or context switching will be necessary since the sequential execution semantics of the SAM enforces that all actions corresponding to a positive section of a session type have always been enqueued by the "caller" process before the "callee" takes over. As discussed in Section 3 it might not seem obvious whether all such input endpoints, (including endpoints moved around via send / receive interactions), always refer to non-empty queues.

Readiness must also be maintained by processes suspended in session records, even if a suspended process waiting on a read endpoint will not necessarily have the corresponding queue already populated. Intuitively, a process P is (H, N)-ready if all its "reads" in the input channels (except those in N) will be matched by values already stored in the corresponding session queue.

Definition 5.2 (Ready). *Process P is H, N-ready if for all $y \in \mathsf{fn}(P) \setminus N$ and $x : A\langle q, R\rangle y \in H$ then A is negative or void. We abbreviate $H, \emptyset$-ready by H-ready. Heap H is ready if, for all $x\langle q, R\rangle y \in H$, the following conditions hold:*

1. *if $R(y)$ then R is $H, \{y\}$-ready*
2. *if $R(x)$ then R is H-ready*
3. *if $\mathsf{clos}(z : -, R) \in q$, R is $H, \{z\}$-ready.*
4. *if $\mathsf{clos}(z : +, R) \in q$, R is H-ready.*

State $C = (P, H)$ is ready if H is ready and P is H-ready.

Lemma 5.1 (Readiness). *Let $P \vdash \emptyset$ and $(P, \emptyset) \overset{*}{\Mapsto} S$. Then $\mathcal{S}$ is ready.*

Theorem 5.3 (Progress). *Let $P \vdash^{\mathsf{B}} \emptyset$ and P live. Then $enc(P) \Mapsto S'$.*

6 The SAM for full CLL

In this section, we complete our initial presentation of the SAM, in particular, we introduce support for the exponentials, allowing the machine to compute with non-linear values, and a selective concurrency semantics. We have delayed the introduction of an environment structure for the SAM, to make the presentation easier to follow. However, this was done at the expense of a more abstract formalisation of the operational semantics, making use of α-conversion, and overloading language syntax names as heap references for allocated session records.

The SAM actually relies on environment-based implementation of name management, presented in Fig. 6. A SAM state is then a triple $(\mathcal{E}, P, H)$ where $\mathcal{E}$ is an environment that maps each free name of the code P into either a closure or a heap record endpoint. These heap references are freshly allocated and unique, thus avoiding any clashes and enforcing proper static scoping. Closures, representing suspended linear $(\mathsf{clos}(z, \mathcal{E}, P))$ and exponential behaviour $(\mathsf{clos!}(z, \mathcal{E}, P))$, pair the code in its environment, and we expect the following structural safety conditions for name biding in configurations to hold.

S	$::= (\mathcal{E}, P, H)$	State
H	$::= Ref \to SessionRec$	Heap
$SessionRec$	$::= x\langle q, \mathcal{E}, P\rangle y$	
q	$::= \mathsf{nil} \mid Val@q$	Queue
Val	$::= \checkmark$	Close token
	$\mid \#\mathsf{l}$	Choice label
	$\mid \mathsf{clos}(x, \mathcal{E}, P)$	Linear Closure
	$\mid \mathsf{clos!}(x, \mathcal{E}, P)$	Exponential Closure
$\mathcal{E}, \mathcal{G}, \mathcal{F}$	$::= Name \to (Ref \cup Val)$	Environment

Fig. 10: The SAM

Definition 6.1 (Closure).
A process P is $(\mathcal{E}, N)$-closed if $\mathsf{fn}(P) \setminus N \subseteq dom(\mathcal{E})$, and $\mathcal{E}$-closed if $(\mathcal{E}, \emptyset)$-closed. Environment $\mathcal{E}$ is H-closed if for all $x \in dom(\mathcal{E})$ if $\mathcal{E}(x)$ is a reference then $x \in H$, if $\mathcal{E}(x) = \mathsf{clos!}(z, \mathcal{F}, R)$ then $\mathcal{F}$ is H-closed and R is $(\mathcal{F}, \{z\})$-closed. Heap H is closed if for all $x\langle q, \mathcal{G}, Q\rangle y \in H$, $\mathcal{G}$ is H-closed, Q is $\mathcal{G}$-closed, and for all $\mathsf{clos}(z, \mathcal{F}, R) \in q$ and $\mathsf{clos!}(z, \mathcal{F}, R) \in q$, $\mathcal{F}$ is H closed and R is $(\mathcal{F}, \{z\})$-closed. State $(\mathcal{E}, P, H)$ is closed if H is closed, $\mathcal{E}$ is H-closed, and P is $\mathcal{E}$-closed.

In Figure 6 we present the environment-based execution rules for the SAM. All rules except those for exponentials have already been essentially presented in Fig. 5 and discussed in previous sections. The only changes to those rules are due to the presence of environments, which at all times record the bindings for free names in the code. Overall, we have

Lemma 6.1. *Let $P \vdash^{\mathsf{B}} \emptyset; \emptyset$. For all S such that $(P, \emptyset, \emptyset) \overset{*}{\Mapsto} S$, S is closed.*

We discuss the SAM rules for the exponentials. Values of exponential type are represented by exponential closures $\mathsf{clos!}(z, \mathcal{F}, R)$. Recall that a session type may terminate in either type $\mathbf{1}$, type $\bot$ or in an exponential type $!A/?A$ (cf. 4.2). So, the (positive) execution rule [S!] is similar to rule [S$\mathbf{1}$]: it enqueues the closure representing the replicated process, and switches context, since the session terminates (cf. [!] Fig. 8). The execution rule [S?] is similar to rule [S$\parr$]: it pops a closure from the queue (which, in this case, always becomes empty), and instead of using it immediately, adds it to the environment to become persistently available to client code (cf. reduction rule [S?] Fig. 8). Any such closure representing a replicated process may be called by client code with transition rule [Scall], which essentially creates a new linear session composed by cut with the client code, similarly to [S$\parr$]. Rule [SCall] operates with some similarity to rule [S$\parr$]: instead of activating a linear closure popped from the queue, it activate an exponential closure fetched from the environment.

We extend the *enc* map to the exponential cut and environment states $(\mathcal{E}, P, H)$ by adapting Definition 5.1, and adding the clause:

$$(\mathcal{E}, \mathsf{cut}\ \{P\ |\overline{x} : A\ [\mathsf{nil}]\ y : B|\ Q\}, H) \mapsto (\mathcal{G}, P, H[a\langle q, \mathcal{F}, Q\rangle b]) \quad [\text{SCut}]$$
$$a, b = \mathsf{new}, \mathcal{G} = \mathcal{E}\{a/x\}, \mathcal{F} = \mathcal{E}\{b/y\}$$

$$(\mathcal{E}, \mathsf{close}\ x, H[a\langle q, \mathcal{F}, P\rangle b]) \mapsto (\mathcal{F}, P, H[a\langle q@\checkmark, \emptyset, 0\rangle b]) \quad [\text{S}\mathbf{1}]$$
$$a = \mathcal{E}(x)$$

$$(\mathcal{E}, \mathsf{fwd}\ x\ y, H[c\langle q_1, \mathcal{G}, Q\rangle a][b\langle q_2, \mathcal{F}, P\rangle d]) \mapsto (\mathcal{F}, P, H[c\langle q_2@q_1, \mathcal{G}, Q\rangle d]) \quad [\text{Sfwd}]$$
$$a = \mathcal{E}(x), b = \mathcal{E}(y)$$

$$(\mathcal{E}, \mathsf{wait}\ y; P, H[a\langle\checkmark, \emptyset, 0\rangle b]) \mapsto (\mathcal{E}, P, H) \quad [\text{S}\bot]$$
$$b = \mathcal{E}(y)$$

$$(\mathcal{E}, \mathcal{A}^-(x), H[a\langle q, \mathcal{G}, Q\rangle b]) \mapsto (\mathcal{G}, Q, H[a\langle q, \mathcal{E}, \mathcal{A}^-(x)\rangle b]) \quad [\text{S}-]$$
$$a = \mathcal{E}(x)$$

$$(\mathcal{E}, \mathsf{send}\ x(z.R); Q, H[a\langle q, P\rangle b]) \mapsto (\mathcal{E}, Q, H[a\langle q@\mathsf{clos}(z, \mathcal{E}, R), P\rangle b]) \quad [\text{S}\otimes]$$
$$a = \mathcal{E}(x)$$

$$(\mathcal{E}, \mathsf{recv}\ y(w{:}+); Q, H[a\langle \mathsf{clos}(z, \mathcal{F}, R)@q, \mathcal{G}, P\rangle b]) \mapsto (\mathcal{E}', Q, H[e\langle \mathsf{nil}, \mathcal{F}', R\rangle f][a\langle q\rangle^s b]) \quad [\text{S}\parr+]$$
$$e, f = \mathsf{new}, b = \mathcal{E}(y), \mathcal{E}' = \mathcal{E}\{e/w\}, \mathcal{F}' = \mathcal{F}\{f/z\}$$

$$(\mathcal{E}, \mathsf{recv}\ y(w{:}-); Q, H[a\langle \mathsf{clos}(z, \mathcal{F}, R)@q, \mathcal{G}, P\rangle b]) \mapsto (\mathcal{F}', R, H[e\langle \mathsf{nil}, \mathcal{E}', Q\rangle f][a\langle q\rangle^s b]) \quad [\text{S}\parr-]$$
$$e, f = \mathsf{new}, b = \mathcal{E}(y), \mathcal{F}' = \mathcal{F}\{e/z\}, \mathcal{E}' = \mathcal{E}\{f/w\}$$

$$(\mathcal{E}, \#\mathsf{l}\ x; Q, H[a\langle q, \mathcal{G}, P\rangle b]) \mapsto (\mathcal{E}, Q, H[a\langle q@\#\mathsf{l}, \mathcal{G}, P\rangle b]) \quad [\text{S}\oplus]$$
$$a = \mathcal{E}(x)$$

$$(\mathcal{E}, \mathsf{case}\ y\ \{|\#\ell \in L{:}Q_\ell\}, H[a\langle \#\mathsf{l}@q, \mathcal{G}, P\rangle b]) \mapsto (\mathcal{E}, Q_{\#\mathsf{l}}, H[a\langle q\rangle^s b]) \quad [\text{S}\&]$$
$$b = \mathcal{E}(y)$$

$$(\mathcal{E}, !x(z); Q, H[a\langle q, \mathcal{G}, P\rangle b]) \mapsto (\mathcal{G}, P, H[a\langle q@\mathsf{clos}(z, \mathcal{E}, Q), \emptyset, 0\rangle b]) \quad [\text{S}!]$$
$$a = \mathcal{E}(x)$$

$$(\mathcal{E}, ?y; Q, H[a\langle \mathsf{clos}(z, \mathcal{F}, R), \emptyset, 0\rangle b]) \mapsto (\mathcal{E}', Q, H) \quad [\text{S}?]$$
$$b = \mathcal{E}(y), \mathcal{E}' = \mathcal{E}\{\mathsf{clos}(z, \mathcal{F}, R)/y\}$$

$$(\mathcal{E}, \mathsf{call}\ y(w{:}+); Q, H) \mapsto (\mathcal{E}', Q, H[a\langle \mathsf{nil}, \mathcal{F}', R\rangle b]) \quad [\text{Scall}+]$$
$$a, b = \mathsf{new}, \mathcal{E}' = \mathcal{E}\{a/w\}, \mathcal{F}' = \mathcal{F}\{b/z\}$$
$$\mathsf{clos}(z, \mathcal{F}, R) = \mathcal{E}(y)$$

$$(\mathcal{E}, \mathsf{call}\ y(w{:}-); Q, H) \mapsto (\mathcal{F}', R, H[a\langle \mathsf{nil}, \mathcal{E}', Q\rangle b]) \quad [\text{Scall-}]$$
$$a, b = \mathsf{new}, \mathcal{E}' = \mathcal{E}\{b/w\}, \mathcal{F}' = \mathcal{F}\{a/z\}$$
$$\mathsf{clos}(z, \mathcal{F}, R) = \mathcal{E}(y)$$

$$a\langle q\rangle^s b \triangleq \text{if } (q = \mathsf{nil}) \text{ then } b\langle q, \mathcal{G}, P\rangle a \text{ else } a\langle q, \mathcal{G}, P\rangle b$$

Fig. 11: SAM Transition Rules for the complete CLL

$$\frac{enc(\mathcal{E}\{\mathsf{clos!}(y,\mathcal{E},R)/x\},P),H) \overset{\mathsf{cut}*}{\mapsto} C}{enc(\mathcal{E},\mathsf{cut!}\ \{y.R\ |!x|\ P\},H) \overset{\mathsf{cut}*}{\mapsto} C}$$

We now update our meta-theoretical results for the complete SAM.

Theorem 6.1 (Soundness). *Let* $P \vdash^{\mathsf{B}} \emptyset;\emptyset$.
If $enc(P) \mapsto D \overset{\mathsf{cut}*}{\mapsto} C$ *then there is* Q *such that* $P \to \cup \equiv Q$ *and* $C = enc(Q)$.

Theorem 6.2 (Progress). *Let* $P \vdash^{\mathsf{B}} \emptyset;\emptyset$ *and* P *live. Then* $enc(P) \mapsto C$.

6.1 Concurrent Semantics of Cut and Mix

Intuitively, the execution of mix $P \parallel Q$ consists in the parallel execution of (non-interfering) processes P and Q. We may execute $P \parallel Q$ by sequentialising P and Q in some arbirary way, and this actually may be useful in some cases.

However, much more interesting is the accommodation in the SAM of interfering concurrency, as required to support full-fledged concurrent languages for session-based programming. First, we evolve the SAM from single threaded to multithreaded, where states now expose a multiset of processes P_i ready for execution by the basic SAM sequential transitions: $(\{P_1, P_2, \ldots, P_n\}, H)$ and introduce an annotated variant pcut of the cut. It has the same CLL/CLLB semantics, but to be implemented as a fork construct where P and Q spawn concurrently, their interaction mediated by an atomic *concurrent session record* $x\,\langle q\rangle\,y$. The type system ensuring that concurrent channels may be forwarded only to concurrent channels. We extend the SAM with transition rule for multisets:

$$\frac{(P,H) \mapsto (P',H')}{(P \uplus T,H) \mapsto (P' \uplus T,H')}\ [\text{Srun}]$$

$$(\mathsf{pcut}\ \{P\ |\overline{x}{:}A\ [q]\ y{:}B|\ Q\} \uplus T,H) \mapsto (\{P,Q\} \uplus T\},H[x\,\langle \mathsf{nil}\rangle\,y])\quad [\text{SCutp}]$$

$$((P\ \parallel\ Q) \uplus T,H) \mapsto (\{P,Q\} \uplus T\},H[x\,\langle \mathsf{nil}\rangle\,y])\quad [\text{SMixp}]$$

Each individual thread executes locally according to the SAM sequential transitions presented before, until an action on a concurrent queue is reached. Concurrent process actions on concurrent queues are atomic, and defined as expected. Positive actions always progress by pushing a value into the queue, while negative actions will either pop off a value from the queue or block, waiting for a value to become available. We illustrate with the rules for $\mathbf{1}$,$\bot$ typed actions.

$$(\mathsf{close}\ x, H[x\,\langle q\rangle\,y]) \mapsto (0, H[x\,\langle q@\checkmark\rangle\,y])\qquad [\text{S}\mathbf{1}\text{c}]$$

$$(\mathsf{wait}\ y; P, H[x\langle \checkmark, y\rangle]) \mapsto (P,H)\qquad [\text{S}\bot\text{c}]$$

Notice that, as in the case for wait $y;P$ above, any negative action in the thread queue is unable to progress if the corresponding queue is empty. It should be clear how to define transition rules for all other pairs of dual actions. Given an appropriate encoding enc^c of annotated CLLB processes in concurrent SAM states, and as consequence of typing and leveraging the proof scheme for progress in CLLB (Theorem 4.2), we have:

Theorem 6.3 (Soundness-c). *Let* $P \vdash^{\mathsf{B}} \emptyset;\emptyset$.
If $enc^c(P) \mapsto D \overset{\mathsf{cut}*}{\mapsto} C$ *then there is* Q *such that* $P \to \cup \equiv Q$ *and* $C = enc^c(Q)$.

Theorem 6.4 (Progress-c). *Let* $P \vdash^{\mathsf{B}} \emptyset;\emptyset$ *and* P *live. Then* $enc_c(P) \Mapsto C$.

The extended SAM executes concurrent session programs, consisting in an arbitrary number of concurrent threads. Each thread deterministically executes sequential code, but can at any moment spawn new concurrent threads. The whole model is expressed in the common language of (classical) linear logic, statically ensuring safety, proper resource usage, termination, and deadlock absence by static typing.

7 Concluding Remarks and Related Work

We introduce the Session Abstract Machine, or SAM, an abstract machine for executing session processes typed by (classical) linear logic CLL, deriving a deterministic, sequential evaluation strategy, where exactly one process is executing at any given point in time. In the SAM, session channels are implemented as single queues with a write and a read endpoint, which are written to, and read by executing processes. Positive actions are non-blocking, giving rise to a degree of asynchrony. However, processes in a session synchronise at polarity inversions, where they alternate execution, according to a fixed co-routining strategy. Despite its specific strategy, the SAM semantics is sound wrt CLL and satisfies the correctness properties of logic-based session type systems. We also present a conservative concurrent extension of the SAM, allowing the degrees of concurrency to be modularly expressed at a fine grain, ranging from fully sequential to fully concurrent execution. Indeed, a practical concern with the SAM design lies in providing a principled foundation for an execution environment for multi-paradigm languages, combining concurrent, imperative and functional programming. The overall SAM design as presented here may be uniformly extended to cover any other polarised language constructs that conservatively extend the PaT paradigm, such as polymorphism, affine types, recursive and co-recursive types, and shared state [56, 61]. We have implemented a SAM-based version [17] of an open-source implementation of CLL [62].

A machine model provides evidence of the algorithmic feasibility of a programming language abstract semantics, and illuminates its operational meaning from certain concrete semantic perspective. Since the seminal work of Landin on the SECD [43], several machines to support the execution of programs for a given programming language have been proposed. The SAM is then proposed herein in this same spirit of Cousineau, Curien and Mauny's Categorical Abstract Machine for the call-by-value λ-calculus [21], Lafont's Linear Abstract Machine for the linear λ-calculus [41], and Krivine's Machine for the call-by-name λ-calculus [40] ; these works explored Curry-Howard correspondences to propose provably correct solutions. In [22], Danvy developed a deconstruction of the SECD based on a sequence of program transformations. The SAM is also derived from Curry-Howard correspondences for linear logic CLL [15, 72], and we also rely on program conversions, via the intermediate buffered language CLLB, as a key proof technique. We believe that the SAM is the first proposal of its kind to tackle the challenges of a process language, while building on several deep properties of its type structure towards a principled design. Among those,

focusing [4] and polarisation [44, 32, 56] played an important role to achieve a deterministic sequential reduction strategy for session-based programming, perhaps our main initial motivation. That allows the SAM to naturally and efficiently integrate the execution of sequential and concurrent session behaviours, and suggests effective compilation schemes for mainstream virtual machines or compiler frameworks.

The adoption of session and linear types is clearly increasing in research (e.g., [26, 3, 58, 24, 74, 66, 56, 61]) and general purpose languages (e.g., Haskell [8, 38], Rust [42, 20] Ocaml [35, 52], F# [51], Move [9], among many others), which either require sophisticated encodings of linear typing via type-level computation or forego of some static correctness properties for usability purposes. Such developments typically have as a main focus the realization of the session typing discipline (or of a particular refinement of such typing), with the underlying concurrent execution model often offloaded to existing language infrastructure.

We highlight the work [19], which studies the relationship between synchronous session types and game semantics, which are fundamentally asynchronous. Their work proposes an encoding of synchronous strategies into asynchronous strategies by so-called call-return protocols. While their focus differs significantly from ours, the encoding via asynchrony is reminiscent of our own.

We further note the work [50] which develops a polarized variant of the $\overline{\lambda}\mu\tilde{\mu}$ suitable for sequent calculi like that of linear logic. While we draw upon similar inspirations in the design of the SAM, there are several key distinctions: the work [50] presents $\lambda\mu$-calculi featuring values and substitution of terms for variables (potentially deep within the term structure). Our system, being based on processes calculus, features neither -– there is no term representing the outcome of a computation, since computation is the interactive behavior of processes (cf. game semantics); nor does computation rely on substitution in the same sense. Another significant distinction is that our work materializes a heap-based abstract machine rather than a stack-based machine. Finally, our type and term structure is not itself polarized. Instead, we draw inspiration from focusing insofar as we extract from focusing the insights that drive execution in the SAM.

In future work, we plan to study the semantics of the SAM in terms of games (and categories), along the lines of [19, 21, 41]. We also plan to investigate the ways in which the evaluation strategy of the SAM can be leveraged to develop efficient compilation of fine-grained session-based programming, and its relationship with effect handlers, coroutines and delimited continuations. Linearity plays a key role in programming languages and environments for smart contracts in distributed ledgers [24, 64] manipulating linear resources (assets); it would be interesting to investigate how linear abstract machines like the SAM would provide a basis for certifying resource sensitive computing infrastructures [75, 9].

Data Availability. An implementation of the SAM as a typechecker and interpreter is publicly available [17]. Additional definitions and proofs can be found in the companion extended technical report [16].

Acknowledgments. This work was supported by NOVA LINCS (UIDB/ 04516/ 2020), INESC ID (UIDB/ 50021/ 2020), BIG (Horizon EU 952226 BIG).

References

1. Abramsky, S.: Computational Interpretations of Linear Logic. Theoret. Comput. Sci. **111**(1–2), 3–57 (1993)
2. Abramsky, S., Gay, S.J., Nagarajan, R.: Interaction categories and the foundations of typed concurrent programming. In: NATO ASI DPD. pp. 35–113 (1996)
3. Almeida, B., Mordido, A., Thiemann, P., Vasconcelos, V.T.: Polymorphic lambda calculus with context-free session types. Inf. Comput. **289**(Part), 104948 (2022)
4. Andreoli, J.M.: Logic Programming with Focusing Proofs in Linear Logic. J. Log. Comput. **2**(3), 297–347 (1992)
5. Balzer, S., Pfenning, F.: Manifest sharing with session types. Proc. ACM Program. Lang. **1**(ICFP) (2017)
6. Bellin, G., Scott, P.: On the π-calculus and linear logic. Theoret. Comput. Sci. **135**(1), 11–65 (1994)
7. Benton, P.N.: A mixed linear and non-linear logic: Proofs, terms and models. In: International Workshop on Computer Science Logic. pp. 121–135. Springer (1994)
8. Bernardy, J., Boespflug, M., Newton, R.R., Jones, S.P., Spiwack, A.: Linear haskell: practical linearity in a higher-order polymorphic language. Proc. ACM Program. Lang. **2**(POPL), 5:1–5:29 (2018)
9. Blackshear, S., Cheng, E., Dill, D.L., Gao, V., Maurer, B., Nowacki, T., Pott, A., Qadeer, S., Russi, D., Sezer, D., Zakian, T., Zhou, R.: Move: A Language with Programmable Resources (2019)
10. Caires, L., Pérez, J.A.: Linearity, control effects, and behavioral types. In: Yang, H. (ed.) Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017. Lecture Notes in Computer Science, vol. 10201, pp. 229–259. Springer (2017)
11. Caires, L., Pérez, J.A.: Linearity, control effects, and behavioral types. In: Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201. p. 229–259. Springer-Verlag, Berlin, Heidelberg (2017)
12. Caires, L., Pérez, J.A., Pfenning, F., Toninho, B.: Behavioral polymorphism and parametricity in session-based communication. In: Proceedings of the 22nd European Conference on Programming Languages and Systems. p. 330–349. ESOP'13, Springer-Verlag, Berlin, Heidelberg (2013)
13. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory. pp. 222–236. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
14. Caires, L., Pfenning, F., Toninho, B.: Towards concurrent type theory. In: Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation. p. 1–12. TLDI '12, Association for Computing Machinery, New York, NY, USA (2012)
15. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Mathematical Structures in Computer Science **26**(3), 367–423 (2016)
16. Caires, L., Toninho, B.: The session abstract machine (extended version) (2024)
17. Caires, L., Toninho, B.: The Session Abstract Machine (Artifact) (2024). https://doi.org/10.5281/zenodo.10459455
18. Cardelli, L.: Typeful Programming. IFIP State-of-the-Art Reports: Formal Description of Programming Concepts pp. 431–507 (1991)
19. Castellan, S., Yoshida, N.: Two sides of the same coin: session types and game semantics: a synchronous side and an asynchronous side. Proc. ACM Program. Lang. **3**(POPL), 27:1–27:29 (2019)

20. Chen, R., Balzer, S., Toninho, B.: Ferrite: A Judgmental Embedding of Session Types in Rust. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming, ECOOP 2022. LIPIcs, vol. 222, pp. 22:1–22:28 (2022)
21. Cousineau, G., Curien, P., Mauny, M.: The Categorical Abstract Machine. Sci. Comput. Program. **8**(2), 173–202 (1987)
22. Danvy, O.: A Rational Deconstruction of Landin's SECD Machine. In: Grelck, C., Huch, F., Michaelson, G., Trinder, P.W. (eds.) Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004. LNCS, vol. 3474, pp. 52–71. Springer (2004)
23. Dardha, O., Gay, S.J.: A new linear logic for deadlock-free session-typed processes. In: Baier, C., Lago, U.D. (eds.) Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018. LNCS, vol. 10803, pp. 91–109. Springer (2018)
24. Das, A., Pfenning, F.: Rast: A language for resource-aware session types. Log. Methods Comput. Sci. **18**(1) (2022)
25. DeYoung, H., Caires, L., Pfenning, F., Toninho, B.: Cut reduction in linear logic as asynchronous session-typed communication. In: Computer Science Logic (2012)
26. Franco, J., Vasconcelos, V.T.: A concurrent programming language with refined session types. In: Counsell, S., Núñez, M. (eds.) Software Engineering and Formal Methods - SEFM 2013. LNCS, vol. 8368, pp. 15–28. Springer (2013)
27. Frumin, D., D'Osualdo, E., van den Heuvel, B., Pérez, J.A.: A bunch of sessions: a propositions-as-sessions interpretation of bunched implications in channel-based concurrency. Proc. ACM Program. Lang. **6**(OOPSLA2), 841–869 (2022)
28. Gay, S., Hole, M.: Subtyping for Session Types in the Pi Calculus. Acta Informatica **42**(2-3), 191–225 (2005)
29. Gay, S., Vasconcelos, V.: Linear Type Theory for Asynchronous Session Types. Journal of Functional Programming **20**(1), 19–50 (2010)
30. Girard, J.: A new constructive logic: Classical logic. Math. Struct. Comput. Sci. **1**(3), 255–296 (1991)
31. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR'93. pp. 509–523. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)
32. Honda, K., Laurent, O.: An exact correspondence between a typed pi-calculus and polarised proof-nets. Theor. Comput. Sci. **411**(22-24), 2223–2238 (2010)
33. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) Programming Languages and Systems. pp. 122–138. Springer (1998)
34. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., et al.: Foundations of Session Types and Behavioural Contracts. ACM Comput. Surv. **49**(1), 3 (2016)
35. Imai, K., Neykova, R., Yoshida, N., Yuen, S.: Multiparty session programming with global protocol combinators. In: Hirschfeld, R., Pape, T. (eds.) 34th European Conference on Object-Oriented Programming, ECOOP 2020. LIPIcs, vol. 166, pp. 9:1–9:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
36. Jacobs, J., Balzer, S.: Higher-order leak and deadlock free locks. Proc. ACM Program. Lang. **7**(POPL), 1027–1057 (2023)
37. Klabnik, S., Nichols, C.: The Rust Programming Language (2021)
38. Kokke, W., Dardha, O.: Deadlock-free session types in linear Haskell. In: Hage, J. (ed.) Haskell 2021: Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell. pp. 1–13. ACM (2021)
39. Kokke, W., Montesi, F., Peressotti, M.: Better late than never: a fully-abstract semantics for classical processes. Proc. ACM Program. Lang. **3**(POPL), 24:1–24:29 (2019)

40. Krivine, J.: A call-by-name Lambda-calculus Machine. High. Order Symb. Comput. **20**(3), 199–207 (2007)
41. Lafont, Y.: The Linear Abstract Machine. Theor. Comput. Sci. **59**, 157–180 (1988)
42. Lagaillardie, N., Neykova, R., Yoshida, N.: Implementing Multiparty Session Types in Rust. In: Coordination Models and Languages Coordination 2020. Lecture Notes in Computer Science, vol. 12134, pp. 127–136. Springer (2020)
43. Landin, P.J.: The Mechanical Evaluation of Expressions. The Computer Journal, Volume 6, Issue 4, January 1964 **6**(4), 308–320 (1964)
44. Laurent, O.: Polarized Proof-Nets: Proof-Nets for LC. In: Girard, J. (ed.) Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99. LNCS, vol. 1581, pp. 213–227. Springer (1999)
45. Lindley, S., Morris, J.G.: Embedding session types in Haskell. In: Mainland, G. (ed.) Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016. pp. 133–145. ACM (2016)
46. Lopes, L.M.B., Silva, F.M.A., Vasconcelos, V.T.: A virtual machine for a process calculus. In: Nadathur, G. (ed.) Principles and Practice of Declarative Programming, International Conference PPDP'99. Lecture Notes in Computer Science, vol. 1702, pp. 244–260. Springer (1999)
47. Milner, R.: Functions as processes. Math. Struct. Comput. Sci. **2**(2), 119–141 (1992)
48. Milner, R.: Elements of interaction: Turing award lecture. Communications of the ACM **36**(1), 78–89 (1993)
49. Milner, R.: Communicating and mobile systems - the Pi-calculus. Cambridge University Press (1999)
50. Munch-Maccagnoni, G.: Focalisation and classical realisability. In: Grädel, E., Kahle, R. (eds.) Computer Science Logic, 23rd international Workshop, CSL 2009. LNCS, vol. 5771, pp. 409–423. Springer (2009)
51. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A session type provider: compile-time API generation of distributed protocols with refinements in f#. In: Dubach, C., Xue, J. (eds.) Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria. pp. 128–138. ACM (2018)
52. Padovani, L.: A simple library implementation of binary sessions. J. Funct. Program. **27**, e4 (2017)
53. Pérez, J.A., Caires, L., Pfenning, F., Toninho, B.: Linear logical relations and observational equivalences for session-based concurrency. Information and Computation **239**, 254–302 (2014)
54. Pfenning, F., Pruiksma, K.: Relating message passing and shared memory, proof-theoretically. In: Jongmans, S., Lopes, A. (eds.) Coordination Models and Languages - COORDINATION 2023. LNCS, vol. 13908, pp. 3–27. Springer (2023)
55. Pfenning, F.: Structural cut elimination. In: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science. p. 156. LICS '95, IEEE Computer Society, USA (1995)
56. Pfenning, F., Griffith, D.: Polarized Substructural Session Types. In: Proc. of FoSSaCS 2015. LNCS, vol. 9034, pp. 3–22. Springer (2015)
57. Pierce, B.C., Turner, D.N.: Pict: a programming language based on the pi-calculus. In: Plotkin, G.D., Stirling, C., Tofte, M. (eds.) Proof, Language, and Interaction, Essays in Honour of Robin Milner. pp. 455–494. The MIT Press (2000)
58. Poças, D., Costa, D., Mordido, A., Vasconcelos, V.T.: System f^{μ} ømega with context-free session types. In: Wies, T. (ed.) Programming Languages and Systems

- 32nd European Symposium on Programming, ESOP 2023. LNCS, vol. 13990, pp. 392–420. Springer (2023)
59. Qian, Z., Kavvos, G., Birkedal, L.: Client-server sessions in linear logic. Proceedings of the ACM on Programming Languages **5**(ICFP), 1–31 (2021)
60. Rocha, P., Caires, L.: Propositions-as-types and Shared State. Proceedings of the ACM on Programming Languages **5**(ICFP), 1–30 (2021)
61. Rocha, P., Caires, L.: Safe session-based concurrency with shared linear state. In: Wies, T. (ed.) Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023. LNCS, vol. 13990, pp. 421–450. Springer (2023)
62. Rocha, P., Caires, L.: Safe session-based concurrency with shared linear state (artifact) (January 2023). https://doi.org/10.5281/zenodo.7506064
63. Sangiorgi, D., Walker, D.: PI-Calculus: A Theory of Mobile Processes. Cambridge University Press, USA (2001)
64. Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., Hao, K.: Safer smart contract programming with Scilla. Proc. ACM Program. Lang. **3**(OOPSLA), 185:1–185:30 (2019)
65. Toninho, B., Caires, L., Pfenning, F.: Functions as Session-Typed Processes. In: FoSSaCS'12. No. 7213 in LNCS (2012)
66. Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: A monadic integration. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. pp. 350–369. Springer (2013)
67. Toninho, B., Caires, L., Pfenning, F.: A decade of dependent session types. In: Veltri, N., Benton, N., Ghilezan, S. (eds.) PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming. pp. 3:1–3:3. ACM (2021)
68. Toninho, B., Yoshida, N.: On polymorphic sessions and functions: A tale of two (fully abstract) encodings. ACM Trans. Program. Lang. Syst. **43**(2) (Jun 2021)
69. Turner, D.N.: The polymorphic Pi-calculus : theory and implementation. Ph.D. thesis, University of Edinburgh, UK (1996)
70. Vasconcelos, V.T.: Lambda and pi calculi, CAM and SECD machines. J. Funct. Program. **15**(1), 101–127 (2005)
71. Wadler, P.: Propositions as sessions. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. p. 273–286. ICFP '12, Association for Computing Machinery, New York, NY, USA (2012)
72. Wadler, P.: Propositions as Sessions. Journal of Functional Programming **24**(2-3), 384–418 (2014)
73. Wadler, P.: Propositions as Types. Commun. ACM **58**(12), 75–84 (2015)
74. Willsey, M., Prabhu, R., Pfenning, F.: Design and implementation of concurrent C0. In: Cervesato, I., Fernández, M. (eds.) Proceedings Fourth International Workshop on Linearity, LINEARITY 2016. EPTCS, vol. 238, pp. 73–82 (2016)
75. Wood, G.: Ethereum: A Secure Decentralised Generalised Transaction Ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)