The Linear Session Abstract Machine

LUÍS CAIRES, Técnico Lisboa and INESC-ID, Portugal

BERNARDO TONINHO, NOVA FCT and NOVA LINCS, Portugal

We introduce the Linear SAM, an abstract machine for mechanically executing session typed programs that precisely correspond to Linear Logic CLL via the propositions-as-types correspondence. In this basic computation model, programs are naturally interpreted as concurrent systems. However, inspired by a fine-grained analysis of proof conversion and focalisation, we derive in this work a fully deterministic sequential evaluation strategy, which may be implemented via co-routining and session buffered communication. Our development targets a language extending CLL with second-order quantifiers (polymorphism) and inductive types (recursion and co-recursion), which supports general higher-order polymorphic functional / session-based computation. A remarkable feature of the SAM's design is its ability to seamlessly coordinate sequential session behaviour with concurrent session behaviour within the same program. We provide an intuitive discussion of the SAM structure and its underlying design, and state and prove its adequacy, showing that SAM executions always correspond to CLL proof reductions, and that any CLL proof reduction is simulated by the SAM execution. To that end, we technically factor our development via an intermediate logical language CLLB, which extends CLL with a "buffered" cut construct, and bridges between the logical/algebraic level and the lower-level machine architecture. We also discuss a proof-of-concept implementation of the SAM, that suggests its potential to support the native linear execution of general session-functional linear programming languages with concurrency.

CCS Concepts: • **Theory of computation** → **Linear logic**; **Type theory**; **Operational semantics**; • **Software and its engineering** → **Functional languages**; **Compilers**; **Semantics**; • **Computing methodologies** → **Concurrent programming languages**.

Additional Key Words and Phrases: Linear Logic, Linear Types, Session Types, Abstract Machine, Semantics

ACM Reference Format:

Luís Caires and Bernardo Toninho. 2024. The Linear Session Abstract Machine. *ACM Trans. Program. Lang. Syst.* 37, 4, Article 111 (May 2024), 56 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In this work, we build on the linear logic based foundation for session types [14, 16, 83] 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, 36, 38, 39], 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 [11, 54, 61, 74, 75, 78, 80], 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

Authors' addresses: Luís Caires, Técnico Lisboa and INESC-ID, Rovisco Pais, 1, Lisbon, Portugal, luis.caires@tecnico.lisboa.pt; Bernardo Toninho, NOVA FCT and NOVA LINCS, Lisboa, Portugal, btoninho@fct.unl.pt.

Manuscript submitted to ACM

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, 2, 7], the proposition-as-types (PaT) interpretation [84] of linear logic proofs as a form of well-behaved session-typed nominal calculus has motivated many developments since its inception [5, 13, 77, 78]. 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 Theory. 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 [53], "data-as-processes" [54] and "functions-as-processes" [52], 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 [20] with linear and behavioral types, e.g., [5, 23, 27, 68]. Most systems of typed session calculi have been formulated in process algebraic form [31, 36, 38], or on top of concurrent λ-calculi with an extra layer of communication channels (e.g., [32]). Logically inspired systems such as those discussed in this paper (e.g., [14, 16, 26, 30, 44, 64, 68, 83]) 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. [62]), with some introducing abstract machines as the underlying technology (e.g., [51, 79]), 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 [24] the LAM [46], or the KM [45], 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" session 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 other 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 [58]. Combining insights related to focalisation and polarisation in linear logic [4, 49, 60], 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 [14, 16, 68], 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" between co-routines 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 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 [42], (Linear) Haskell [9, 43, 50], Move [10], and in research languages [27, 41, 66]. To further substantiate these views we have developed a proof-of-concept implementation of the SAM, integrated as an alternative backend for CLASS [68], an experimental language for session-based programs [18].

1.1 Outline and Contributions

In Section 2 we briefly review the session-typed calculus CLL, which exactly corresponds to (classical) Linear Logic with mix, second-order quantifiers, and (co)inductive types. 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 and polymorphism, which will be introduced later. Even if the core SAM structure and transition rules are fairly simple, the required proofs of correctness are more technically involved, and require progressive build up. Therefore, in Section 4 we first bridge between CLL and SAM via an intermediate logical language CLLB, which introduces cuts with explicit queues. We show preservation (Theorem 4.8) and progress (Theorem 4.10) for CLLB, and prove that there is a two way simulation between CLLB and CLL via a strong operational correspondence (Theorem 4.15).

Given this correspondence, in Section 5 we state and prove the adequacy of the SAM for executing CLL processes, showing soundness wrt. CLLB (Theorem 5.6) and CLL (Theorem 5.7), and progress / deadlock absense (Theorem 5.8). In Section 6 we modularly extend the previous results to the exponentials, and revise the core SAM by introducing explicit environments, stating the associated adequacy results (Theorem 6.7 and Theorem 6.8).

In Section 7 we discuss a uniform and modular extension of the SAM towards concurrency, and present the related correctness results. This demonstrates how the SAM design naturally allows sequential and concurrent session behaviours to be smoothly coordinated, while preserving the basic safety properties of the type system, namely soundness (Theorem 7.5) and progress / deadlock-freedom (Theorem 7.6). In Section 8 we briefly describe a proof-of-concept implementation of the SAM and discuss its potential to support the native linear execution of general linear programming languages with concurrency. We illustrate with some examples, including a lazy stream-based prime sieve, System-F style encoding of recursive types, Ackermann computation on recursively defined natural numbers, and an infinite precision binary counter. We conclude in Section 9 by a discussion of related work and additional remarks.

2 THE CLL LANGUAGE AND ITS 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 Girard's linear logic [34]; in this work we follow standard notations for classical linear logic proofs as processes [12, 16, 66, 68, 83], where types correspond to propositions and language constructs to proof rules. We start by types and duality.

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

$$\begin{array}{llllllllll} A, B ::= & \mathbf{1} & \mid \bot & \mid A \parr B & \mid A \otimes B & \mid \&_{\ell \in L}\, A_\ell & \mid \oplus_{\ell \in L}\, A_\ell & \mid\, !A & \mid\, ?A \mid \\ & X & \mid \overline{X} & \mid \exists X.A & \mid \forall X.A & \mid \mu X.A & \mid \nu X.A & & \end{array}$$

Types comprise of the units ($\mathbf{1}$, $\bot$), multiplicatives ($\otimes$, $\parr$), additives ($\oplus_{\ell \in L} A_\ell$, $\&_{\ell \in L} A_\ell$), exponentials ($!$, $?$), type variables, (X, Y) and dual type variables ($\overline{X}$, $\overline{Y}$), quantifiers ($\exists$, $\forall$) and inductive types (μ, ν). We adopt here an applied

$$
\begin{array}{lllllll}
P, Q & ::= & \mathsf{0} & & & & \\
 & \mid & P \mid\mid Q & & \mid & !x(y);P & !A \\
 & \mid & \mathsf{fwd}\ x\ y & & \mid & ?x;P & ?A \\
 & \mid & \mathsf{cut}\ \{P \mid x{:}A \mid Q\} & & \mid & \mathsf{cut!}\ \{y.P \mid !x : A \mid Q\} & \\
 & \mid & \mathsf{close}\ x & \mathbf{1} & \mid & \mathsf{call}\ x(z);Q & \\
 & \mid & \mathsf{wait}\ x;P & \bot & \mid & \mathsf{sendty}\ x(B);P & \exists X.A \\
 & \mid & \mathsf{case}\ x\ \{\mid \#\ell \in L{:}P_\ell\} & \&_{\ell \in L} A_\ell & \mid & \mathsf{recvty}\ x(X);P & \forall X.A \\
 & \mid & \#\mathsf{l}\ x;P & \oplus_{\ell \in L} A_\ell & \mid & \mathsf{unfold}_\mu\ x;P & \mu X.A \\
 & \mid & \mathsf{send}\ x(y.P);Q & A \otimes B & \mid & \mathsf{rec}\ X(z,\vec{w});P\ [x,\vec{y}] & \nu X.A \\
 & \mid & \mathsf{recv}\ x(z);P & A \parr B & \mid & \mathsf{unfold}_\nu\ x;P & \nu X.A
\end{array}
$$

Fig. 1. Program terms (processes) of CLL, and hint to type assigned to subject channel x

version of the additive types, where e.g. the primitive linear logic (binary) sum type $A \oplus B$ is replaced by a finite collection of labeled options $\oplus_{\ell \in L} A_\ell$. There is notion of polarity for types [49] where the *positive types* are $\mathbf{1}$, $\otimes$, $\oplus$, and $!$, and the *negative types* are $\bot$, $\parr$, $\&$ and $?$. We write A^+ (resp. A^-) to assert that A is a positive (resp. negative) type.

Definition 2.2 (Duality). Type *duality* $A \mapsto \overline{A}$ is the involution on types induced by linear logic negation:

$$
\begin{array}{lllllllllll}
\overline{X} & = & \overline{X} & \overline{\mathbf{1}} & = & \bot & \overline{A \otimes B} & = & \overline{A} \parr \overline{B} & \overline{\oplus_{\ell \in L} A_\ell} & = & \&_{\ell \in L} \overline{A_\ell} \\
\overline{\overline{X}} & = & X & \overline{!A} & = & ?\overline{B} & \overline{\exists X.A} & = & \forall X.\overline{A} & \overline{\mu X.A} & = & \nu X.\overline{A}
\end{array}
$$

Duality captures the symmetry of behaviour in binary process interaction, as manifest in the cut rule. We may abbreviate $\overline{A} \parr B$ by $A \multimap B$. We denote by $\{T/X\}A$ the capture-free substitution of type X for variable X in type A. Notice that $\{T/X\}\overline{A} = \overline{\{T/X\}A}$.

Definition 2.3 (Processes). The syntax of CLL program terms (processes) P, Q is defined in Figure 1.

The typing rules of CLL are presented in Figures 2, 3 and 4. Typing judgements have the form $P \vdash_\eta \Delta; \Gamma$, where P is a process and the typing context $\Delta; \Gamma$ is dyadic [4, 8, 14, 59]: both Δ and Γ assign types to names, the context Δ is handled linearly while the exponential context Γ is unrestricted. This means that no implicit contraction or weakening is allowed on Δ. The type system exactly corresponds, via a propositions-as-types [14, 16, 82] correspondence, to second-order classical linear logic with mix, extended with inductive/coinductive types (cf. [65, 68, 76]). The index η is a finite map from type variables to coinduction judgements, used in typing rules for corecursive types, further discussed below.

2.1 Identity - Mix, Cut, Inaction

The process $\mathsf{0}$ denotes the inaction (do-nothing, terminated) process, typed in the empty linear context (rule [T$\mathsf{0}$]).

The mix $P \mid\mid Q$ denotes independent parallel composition of processes P and Q (rule [Tmix]), whereas the $\mathsf{cut}\ \{P \mid x{:}A \mid Q\}$ denotes parallel composition of communicating processes P and Q, where P and Q share exactly one channel x, typed as A in P and $\overline{A}$ in Q (rule [Tcut]). In many situations, a cut type annotation may be easily inferred from the context, in such cases we may omit it to light notation, and write $\mathsf{cut}\ \{P \mid x \mid Q\}$.

The forwarder $\mathsf{fwd}\ x\ y$ captures bidirectional forwarding between dually typed names x and y (rule [Tfwd]), which operationally consists in renaming x for y.

$$\frac{}{\mathsf{0} \vdash \emptyset; \Gamma}\;[\mathrm{T}\mathsf{0}] \qquad \frac{P \vdash \Delta'; \Gamma \quad Q \vdash \Delta; \Gamma}{P \mathbin{||} Q \;\vdash \Delta', \Delta; \Gamma}\;[\mathrm{Tmix}]$$

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

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

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

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

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

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

Fig. 2. Typing Rules I: CLL.

$$\frac{P \vdash \Delta, x : \{B/X\}A; \Gamma}{\mathsf{sendty}\; x(B); P \;\vdash \Delta, x : \exists X.A; \Gamma}\;[\mathrm{T}\exists] \qquad \frac{Q \vdash \Delta, x : A; \Gamma}{\mathsf{recvty}\; x(X); Q \vdash \Delta, x : \forall X.A; \Gamma}\;[\mathrm{T}\forall]$$

Fig. 3. Typing Rules II: Polymorphism.

$$\frac{P \vdash_{\eta'} \Delta, z : A; \Gamma \quad \eta' = \eta, X(z, \vec{w}) \mapsto \Delta, z : Y; \Gamma}{\mathsf{rec}\; X(z, \vec{w}); P\; [x, \vec{y}] \vdash_\eta \{\vec{y}/\vec{w}\}\Delta, x : \nu Y.A; \{\vec{y}/\vec{w}\}\Gamma}\;[\mathrm{Tcorec}] \qquad \frac{\eta = \eta', X(x, \vec{y}) \;\mapsto \Delta, x : Y; \Gamma}{X(z, \vec{w}) \vdash_\eta \{\vec{w}/\vec{y}\}\Delta, z : Y; \{\vec{w}/\vec{y}\}\Gamma}\;[\mathrm{Tvar}]$$

$$\frac{P \vdash_\eta \Delta, x : \{\mu X.A/X\}A; \Gamma}{\mathsf{unfold}_\mu\; x; P \vdash_\eta \Delta, x : \mu X.A; \Gamma}\;[\mathrm{T}\mu]$$

Fig. 4. Typing Rules III: Induction and Coinduction.

2.2 Multiplicatives - Send, Receive, Close, Wait

Process close x explicitly initiates termination of the session, while wait $x; P$ represents the dual action of waiting for session termination, as typed by rules [T$\mathbf{1}$] and [T$\bot$].

Process send $x(y.P_1); P_2$ outputs on channel x a fresh channel y performing a behaviour specified by P_1, and continues as P_2 afterwards (rule [T$\otimes$]). The process recv $x(z); Q$ receives from x a channel on input parameter z and continues as Q, which will have access and use z (rule [T$\parr$]). When such send and receive processes interact on dual endpoints of a session x, the behaviour defined by P_1 is linearly passed from sender to receiver, via a dual action synchronisation.

Notice that the sent name y is bound in $\mathsf{send}\ x(y.P_1); P_2$ with scope P_1, and the input parameter name z is bound in $\mathsf{recv}\ x(z); Q$ with scope Q.

2.3 Additives

Process $\#\mathsf{l}\ x; P$ selects label $\#\mathsf{l}$ on session x and continues as process P. It is typed by a labeled sum type of the form $\oplus_{\ell \in L} A_\ell$ (rule [T$\oplus$]). Such label selection always acts on a offer process $\mathsf{case}\ x\ \{|\#\ell \in L{:}P_\ell\}$ holding the dual endpoint of session x. The offer process branches to continuation P_ℓ depending if the selected label is $\#\ell$, and would be typed by the offer type $\&_{\ell \in L} \overline{A}_\ell$ (rule [T$\&$]).

2.4 Exponentials

Processes $!x(y); P$, $?x; Q$ and $\mathsf{call}\ x(z); Q$ embody replicated processes and their invocations. Replicated processes represent non-linear computations, comparable to exponential (or intuitionistic) values as available in functional languages, that may be dropped or used an arbitrary number of times.

Process $!x(y); P$ represents of a process that may be called at x, to yield a fresh new behavior at y, as defined by 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 $\mathsf{cut!}\ \{y.P\ |!x : A|\ Q\}$ process, where P cannot depend on linear sessions and so may be safely replicated.

2.5 Quantifiers - Polymorphism

The process $\mathsf{sendty}\ x(T); P$ sends a type T on along x and continues as P and is assigned type $\exists X.A$ (rule [T$\exists$]). The matching receiver process $\mathsf{recvty}\ x(X); Q$ receives the sent type on x, instantiating type input parameter X in Q, which provides the continuation. The type parametric receiver is assigned type $\forall X.\overline{A}$. Existential and universal types allow us to introduce type abstraction and parametricity in our session typed language [13, 68, 82, 83]. In particular, the resulting language has the same expressive power as Linear System F [78].

2.6 Induction and Coinduction - Recursion

We follow the presentation of inductive / coinductive session types developed for classical linear logic [65, 68], which in turn built on the intuitionistic version [72, 76]. Corecursive processes introduced by rule [Tcorec] have the form $\mathsf{rec}\ X(z, \vec{w}); P\ [x, \vec{y}]$. In such a process, the subterm $\mathsf{rec}\ X(z, \vec{w}); P$ denotes a (co) recursively defined parametric process abstraction where the parameters $z, \vec{w}$ are bound in the body P, where they may be used. The process variable X is also bound in P, and occurs free in P to express recursive calls, each of the form $X(a, \vec{b})$ for some parameters $(a, \vec{b})$. In the whole process $\mathsf{rec}\ X(z, \vec{w}); P\ [x, \vec{y}]$, the free names $x, \vec{y}$ are passed as arguments to the current call on the recursive process. The coinductive behaviour of type $\nu Y.A$ of a corecursive process is always offered by the first argument channel z. In rule [Tcorec] to type the body P of a corecursive process, the map η is extended with a coinductive hypothesis that binds the process variable X to the current typing context $\Delta, z : Y; \Gamma$. Whenever a call to X appears inside the body P we rely on $\eta(X)$ to recover the appropriate co-inductive invariant. This is captured by rule [Tvar], that types a corecursive call $X(z, \vec{w})$ by looking up in η for the corresponding binding and renaming the parameters with the arguments of the call. Inductive and coinductive types are explicitly unfolded with rule [Tμ].

$$\begin{array}{lll}
\mathsf{fwd}\ x\ y\ \equiv \mathsf{fwd}\ y\ x & [\text{fwd}] & (\text{provisos}) \\
\mathsf{cut}\ \{P\ |x : A|\ Q\}\ \equiv \mathsf{cut}\ \{Q\ |x : \overline{A}|\ P\} & [\text{com}] & \\
P \parallel 0\ \equiv P \quad P \parallel Q \equiv Q \parallel P \quad P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R & [\text{par}] & \\
\mathsf{cut}\ \{P\ |x|\ (Q \parallel R)\} \equiv (\mathsf{cut}\ \{P\ |x|\ Q\}) \parallel R & [\text{CM}] & x \in \mathsf{fn}(Q) \\
\mathsf{cut}\ \{P\ |x|\ (\mathsf{cut}\ \{Q\ |y|\ R\})\} \equiv^{\mathsf{B}} \mathsf{cut}\ \{Q\ |y|\ (\mathsf{cut}\ \{P\ |x|\ R\})\} & [\text{CC}] & x, y \in \mathsf{fn}(R) \\
\mathsf{cut}\ \{P\ |z|\ (\mathsf{cut!}\ \{y.Q\ |!x|\ R\})\} \equiv \mathsf{cut!}\ \{y.Q\ |!x|\ (\mathsf{cut}\ \{P\ |z|\ R\})\} & [\text{CC!}] & x \notin \mathsf{fn}(Q) \text{ and } z \notin \mathsf{fn}(P) \\
\mathsf{cut!}\ \{y.Q\ |!x|\ (P \parallel R)\} \equiv P \parallel (\mathsf{cut!}\ \{y.Q\ |!x|\ R\}) & [\text{C!M}] & x \notin \mathsf{fn}(Q) \text{ and } z \notin \mathsf{fn}(P) \\
\mathsf{cut!}\ \{y.P\ |!x|\ (\mathsf{cut!}\ \{w.Q\ |!z|\ R\})\} \equiv \mathsf{cut!}\ \{w.Q\ |!z|\ (\mathsf{cut!}\ \{y.P\ |!x|\ R\})\} & [\text{C!C!}] & \\
\mathsf{cut!}\ \{y.P\ |!x|\ (Q \mid * \mid R)\} \equiv \mathsf{cut!}\ \{y.P\ |!x|\ Q\} \mid * \mid \mathsf{cut!}\ \{y.P\ |!x|\ R\} & [\text{C!*}] & \\
a(x); Q \mid * \mid R\ \equiv a(x); (Q \mid * \mid R) & [\text{C+*}] & \\
a(x); b(y); P\ \equiv b(y); a(x); P & [\text{C}i] & x \neq y,\ bn(a(x)) \cap fn(b(y)) = \emptyset
\end{array}$$

Fig. 5. Structural congruence $P \equiv Q$.

2.7 Reduction Semantics

We call *action* any process that either realizes an introduction rule ($[T\otimes]$, $[T\parr]$, $[T\mathbf{1}]$, $[T\bot]$, $[T!]$, $[T?]$) or is a forwarder. We then 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 [54]). An action is deemed *positive* (resp. *negative*) if its associated type is positive (resp. negative) in the sense of focusing [4] and polarisation [49]. The set of positive (resp. 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 $\longrightarrow$ that captures process interaction, and corresponds to cut-elimination steps.

Definition 2.4 ($P \equiv Q$). Structural congruence $\equiv$ is the least congruence on processes closed under α-conversion and the $\equiv$-rules in Fig. 5.

The definition of $\equiv$ reflects expected static laws, along the lines of the structural congruences / conversions in [14, 82]. 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 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 $- \mid * \mid -$ 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 [58]. These conversions are not required to obtain deadlock freedom. However, they are necessary for full cut elimination (e.g., see [82]), and expose more redexes, thus more non-determinism in the choice of possible reductions. Perhaps surprisingly, this extra flexibility is important to allow the deterministic sequential evaluation strategy for CLL programs adopted by the SAM to be expressed.

Definition 2.5 (Reduction $\longrightarrow$). Reduction $\longrightarrow$ is defined by the rules of Figure 6.

For readability, we omit the type declarations in the cuts, as they do not actually play any operational role in reduction. We denote by $\Rightarrow$ the reflexive-transitive closure of $\longrightarrow$. Reduction includes the set of principal cut conversions, i.e. the

$$
\begin{array}{ll}
\mathsf{cut}\ \{\mathsf{fwd}\ x\ y\ |y|\ P\} \longrightarrow \{x/y\}P & [\mathrm{fwd}] \\
\mathsf{cut}\ \{\mathsf{close}\ x\ |x|\ \mathsf{wait}\ x; P\} \longrightarrow P & [\mathbf{1}\bot] \\
\mathsf{cut}\ \{\mathsf{send}\ x(y.P); Q\ |x|\ \mathsf{recv}\ x(z); R\} \longrightarrow \mathsf{cut}\ \{Q\ |x|\ (\mathsf{cut}\ \{P\ |y|\ \{y/z\}R\})\} & [\otimes\invamp] \\
\mathsf{cut}\ \{\mathsf{case}\ x\ \{|\#\ell \in L{:}P_{\#\ell}\}\ |x|\ \#\mathsf{l}\ x; R\} \longrightarrow \mathsf{cut}\ \{P_{\#\mathsf{l}}\ |x|\ R\} & [\&\oplus_l] \\
\mathsf{cut}\ \{!x(y); P\ |x|\ ?x; Q\} \longrightarrow \mathsf{cut!}\ \{y.P\ |!x|\ Q\} & [!?] \\
\mathsf{cut!}\ \{y.P\ |!x|\ \mathsf{call}\ x(z); Q\} \longrightarrow \mathsf{cut}\ \{\{z/y\}P\ |z|\ (\mathsf{cut!}\ \{y.P\ |!x|\ Q\})\} & [\mathrm{call}] \\
\mathsf{cut}\ \{\mathsf{sendty}\ x(A); P\ |x|\ \mathsf{recvty}\ x(X); Q\} \longrightarrow \mathsf{cut}\ \{P\ |x|\ \{A/X\}Q\} & [\exists\forall] \\
\mathsf{cut}\ \{\mathsf{unfold}_\mu\ x; P\ |x|\ \mathsf{rec}\ Y(z,\vec{w}); Q\ [x,\vec{y}]\} \longrightarrow \mathsf{cut}\ \{P\ |x|\ \{x/z\}\{\vec{y}/\vec{w}\}\{\mathsf{rec}\ Y(z,\vec{w}); Q/Y\}Q\} & [\mathrm{corec}]
\end{array}
$$

Fig. 6. Reduction $P \to Q$.

redexes for each pair of interacting constructs. It is closed by structural congruence ([≡]), in rule [cong] we consider that 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] [15]. 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 the basic safety properties, type preservation and progress. Preservation follows since structural congruence and reductions are sound linear logic proof conversions (see [14, 16, 68]).

Theorem 2.6 (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 $P = C[Q]$, for some static context C (the hole lies within the scope of static constructs mix and cut) and where Q is an action process. The proof of Theorem 2.7 (see [14, 16, 68]) leverages deep properties of linear logic proofs, allowing deadlock absence to be proved by purely structural means.

Theorem 2.7 (Progress). *Let $P \vdash \emptyset; \emptyset$ be live. Then $P \to Q$ for some Q.*

2.8 Examples

After formally presenting the operational semantics of our language, we illustrate the various operators with some example code written in a programmer friendly, sugared syntax.

To simplify the presentation of examples, we omit explicit unfolding actions, and write inductive and coinductive type definitions with equations of the form $\mathsf{rec}\ A = f(A)$ and $\mathsf{corec}\ B = f(B)$ instead of $A = \mu X.f(X)$ and $B = \nu X.f(X)$, respectively. Similarly, we write corecursive process definitions as $Q(x,\vec{y}) = f(Q(-))$ instead of $Q(x,\vec{y}) = \mathsf{rec}\ X(z,\vec{w}); f(X(-))\ [x,\vec{y}]$, while of course respecting the constraints imposed by typing rules [Tvar] and [Tcorec].

We begin by defining a recursive session type encoding the natural numbers:

$$\mathsf{rec}\ \mathsf{Nat} = \oplus\{\,\#\mathsf{Z} : \mathbf{1}, \#\mathsf{S} : \mathsf{Nat}\,\}$$

We can now define processes implementing some natural numbers incrementally as follows:

$$\begin{aligned}
\mathsf{zero}(n : \mathsf{Nat}) &= \#\mathsf{Z}\ n; \mathsf{close}\ x\\
\mathsf{one}(n : \mathsf{Nat}) &= \#\mathsf{S}\ n; \mathsf{zero}(n)\\
\mathsf{two}(n : \mathsf{Nat}) &= \#\mathsf{S}\ n; \mathsf{one}(n)
\end{aligned}$$

We can now define a recursive process which duplicates a given natural number (on channel n) by emitting the corresponding behavior on channel r:

$$\begin{aligned}
\mathsf{dupl}(n : \overline{\mathsf{Nat}}, r : \mathsf{Nat}) = \mathsf{case}\ \{\ &\#\mathsf{Z} : \mathsf{wait}\ n; \#\mathsf{Z}\ r; \mathsf{close}\ r,\\
&\#\mathsf{S} : \#\mathsf{S}\ r; \#\mathsf{S}\ r; \mathsf{dupl}(n, r)\ \}
\end{aligned}$$

Process $\mathsf{dupl}(n, r)$ consumes a (linear) Nat at n and produces a (linear) Nat at r. It performs case analysis on the label sent along channel n. If the label corresponds to zero ($\#Z$), then the corresponding label is sent along r. If the label corresponds to a successor ($\#S$), then two successor labels are sent along r, and then the process recurs on n, thus doubling the value of n on r. We can define a replicable (non-linear) version of the doubling process via the exponentials as follows:

$$\mathsf{ddup}(dup : !(Nat \multimap Nat)) = !dup(f); \mathsf{recv}\ f(x); \mathsf{dupl}(x, f)$$

The ddup process above provides at dup a replicated "function" process, which may be called on channel dup to yield a fresh linear session (on fresh channel f) that will input the number x to be doubled using dupl. Notice that the type $!(Nat \multimap Nat)$ abbreviates $!(\overline{Nat} \parr Nat)$. A shared usage of such a function process by two parallel clients, one calling with the number one and the other with the number two, is given by program main below.

$$\begin{array}{ll}
\mathsf{main}() = & \mathsf{cut}\ \{\\
& \quad \mathsf{ddup}(dup)\\
& \quad |dup : !(Nat \multimap Nat)|\\
& \quad (\\
& \quad \mathsf{call}\ dup(c_1); \mathsf{send}\ c_1(n.\mathsf{one}(n)); \mathsf{printnat}(c_1)\\
& \quad ||\\
& \quad \mathsf{call}\ dup(c_2); \mathsf{send}\ c_2(n.\mathsf{two}(n)); \mathsf{printnat}(c_2)\\
& \quad)\\
& \}
\end{array}$$

$$\begin{aligned}
&\mathsf{rec}\ \mathsf{printnat}(n : \overline{\mathsf{Nat}}) =\\
&\qquad \mathsf{case}\ n\ \{\ \#\mathsf{Z} : \mathsf{wait}\ n; 0,\\
&\qquad\qquad \#\mathsf{S} : \mathsf{printnat}(n)\ \}
\end{aligned}$$

where $\mathsf{printnat}(n)$ simply consumes down the given natural number n.

3 THE DESIGN OF THE SAM

In this section we develop the key insights that guide the construction of our linear session abstract machine (SAM) and introduce its operational rules in an incremental fashion. We consider here the basic (multiplicative / additive) fragment of linear logic for the sake of clarity of presentation, postponing the analysis of exponentials, polymorphism and recursion to 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*

$$
\begin{array}{llll}
S & ::= & (P, H) & \text{Configuration} \\
R & ::= & x, y & \text{SRef} \\
H & ::= & (\mathit{SRef}, \mathit{SRef}) \rightarrow \mathit{SessionRec} & \text{Heap} \\
R & ::= & x\langle q, P\rangle y & \text{Session Record} \\
q & ::= & \mathsf{nil} \mid V \mid V@q & \text{Queue} \\
\mathit{Val} & ::= & \checkmark & \text{Close token} \\
 & \mid & \#\mathsf{l} & \text{Choice label} \\
 & \mid & \mathsf{clos}(x, P) & \text{Process Closure}
\end{array}
$$

Fig. 7. The Core SAM Components

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 does not prescribe a rigid evaluation order for processes. For instance, in the process $\mathsf{cut}\ \{P\ |x|\ Q\}$, reduction is allowed in both P and Q. This is of course in line with reduction in process calculi (e.g., [54]). 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 [28]), 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 on the same session.

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 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 13. An execution state is simply a pair consisting of *the* running process P and the heap H.

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 #l or a process closure $\mathsf{clos}(x, P)$, used to model session send and receive. While omitted for the sake of clarity at this stage, in Section 5 we expand the grammar of queue contents to include types $\mathsf{ty}(T)$ (exchanged by polymorphic processes) and recursion markers step, to account for recursion.

Cut. We begin by considering how to execute a cut of the form $\mathsf{cut}\ \{P\ |x : A|\ Q\}$, which consists of the composition of processes P and Q, exclusively sharing the new session channel x. Given the choice of either scheduling P or Q, we rely on the *polarity* (in the sense of polarized logic [33]) of type A to drive the execution of the SAM. A positive type corresponds to a type denoting an *output* (or write) action, whereas a negative type denotes an *input* (or read) action. We are thus forced to schedule the process whose next action on x is a *write* rather than a *read*: the only way for a process to exercise its read capability on such a new session successfully is to wait for the writer to have exercised (at least some of) its write capability.

Recalling that P uses x according to type A and Q according to $\overline{A}$, if A is positive, we must schedule P. If A is negative, we must schedule Q. Thus, the SAM rule for cut is:

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

The rule allocates a new session record with an empty queue, relying on the operation $(P(x), H[x{:}A\langle \mathsf{nil}, Q(y)\rangle y{:}\overline{A}])^{p}$ which used to prepare the newly created session. The operation, defined as

$$(P, H[x{:}A\langle \mathsf{nil}, Q\rangle y{:}B])^{p} \triangleq \text{if } (A+) \text{ then } (P, H[x{:}A\langle \mathsf{nil}, Q\rangle y{:}B]) \text{ else } (Q, H[y{:}B\langle \mathsf{nil}, P\rangle x{:}A])$$

essentially schedules process $P(x)$ for execution if the type A of x is positive, which then becomes the write endpoint, and suspends $Q(y)$ at the negative endpoint y. If A is negative, the session record is set conversely, with $Q(y)$ scheduled and $P(x)$ suspended. Note that in general, the holder of the write and read endpoint can change throughout execution of the machine. Moreover, 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 lookup the session record for x and add to the queue a *process closure* containing R (which interacts along z). We must then choose the next process to execute. Again, this choice relies on the polarity of the (continuation) session type. If the type is positive, execution will continue with the continuation process Q. Otherwise, execution will switch control to the suspend process P, holding the read endpoint of the session:

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

The control choice is implemented using the operation $(P, H[x{:}A\langle q, Q\rangle y{:}B])^{wr}$ (write-to-read adjust), given by:

$$(P, H[x{:}A\langle q, Q\rangle y{:}B])^{wr} \triangleq \text{if } (A+) \text{ then } (P, H[x{:}A\langle q, Q\rangle y : B]) \text{ else } (Q, H[x{:}A\langle q, P\rangle y{:}B])$$

The write-to-read adjust operation prepares the state of the SAM after a positive operation in an ongoing session, according to the description above, effectively allowing all positive operations in sequence in a given session to take place before a context switch to the reader process happens (once the type polarity switches from positive to negative).

Session Closure. The execution of close follows a similar spirit, but no continuation process exists in this case and the SAM resumes the process P holding the *read* endpoint y of the queue:

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

The process P will eventually read, via wait, the session termination mark from the queue, triggering the deallocation of the session record from the heap:

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

Note the requirement that ✓ be the final element of the queue.

Channel Input. The rule for recv is as follows:

$$(\mathsf{recv}\ y(w : \overline{A}); Q, H[x{:}C\langle \mathsf{clos}(z : A, R)@q, P\rangle y{:}\overline{A} \parr D]) \Mapsto (Q, H[(x{:}C\langle q, P\rangle y{:}D)^{rw}][w{:}\overline{A}\langle \mathsf{nil}, R\rangle z{:}A])^{p} \qquad [\mathrm{S}\parr]$$

where $(x{:}A\langle q, Q\rangle y{:}B)^{rw} \triangleq$ if $(q = \mathsf{nil})$ then $y{:}B\langle q, Q\rangle x{:}A$ else $x{:}A\langle q, Q\rangle y{:}B$. 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 rely once again on the init-adjust operation $(\ldots)^{p}$, which schedules R or Q depending on the polarity of the type of w. If $\overline{A}$ is positive, Q will eventually write on w, thus w is the positive or write endpoint and Q is scheduled for execution.

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 is achieved by the read-to-write adjust operation $(x{:}A\langle q, Q\rangle y{:}B)^{rw}$. This swap serves two purposes: first, it enables Q to perform writes if needed; secondly, and more subtly, it allows for the process 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 just before 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 in a negative polarity action which will context switch to P through the write-to-read adjust operation.

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

$$(\#\mathsf{l}\ x; Q, H[x : \oplus_{\ell\in L} A_{\ell}\langle q, P\rangle y : B]) \Mapsto (Q, H[x : A_{\#\mathsf{l}}\langle q@\#\mathsf{l}, P\rangle y : B])^{wr} \qquad [\mathrm{S}\oplus]$$

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

Sending a label #l, a positive action, simply adds the #l to the corresponding queue and proceeds with the execution, performing the write-to-read adjustment analogously to the [S⊗ rule. 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 read-to-write adjustment as in rule [S⅋].

Forwarding. Finally, let us consider the execution of a forwarder:

$$(\mathsf{fwd}\ x\ y, H[z : A\langle q_1, Q\rangle x : \overline{B}][y : B\langle q_2, P\rangle w : C]) \Mapsto (P, H[z : A\langle q_2@q_1, Q\rangle w : C]) \qquad [\mathrm{Sfwd}]$$

A forwarder denotes the merging of two different sessions x and y, with $x : \overline{B}$ and $y : B$. Without loss of generality (because of $\mathsf{fwd}\ x\ y \equiv \mathsf{fwd}\ y\ x$), we assume the forwarder $\mathsf{fwd}\ x\ y$ holds the read and write endpoints at respectively x and y, with $\overline{B}$ negative and B positive. 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. These cases are *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).

3.1 On the Write-Bias of the SAM

Consider the following CLL process:

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

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

The execution of P in the SAM begins in the state:

$$(P, \emptyset)$$

Which executes the cut through rule [SCut]. Since the type of a is positive, we execute P_1, and allocate the session record, suspending Q_1:

$$(P, \emptyset) \Mapsto (P_1, a\langle \mathsf{nil}, Q_1\rangle b)$$

Since P_1 is a write action on a write endpoint, we proceed via the [S$\otimes$] rule, resulting in the SAM configuration,

$$(P_1, a\langle \mathsf{nil}, Q_1\rangle b) \Mapsto (P_3, a\langle \mathsf{clos}(y, P_2), Q_1\rangle b)$$

executing P_3 and adding a closure containing P_2 to the session queue with write endpoint a. To execute P_3, a close action, we add the ✓ to the queue and switch to the process Q_1 (rule [S$\mathbf{1}$]), now ready to receive the sent value:

$$(P_3, a\langle \mathsf{clos}(y, P_2), Q_1\rangle b) \Mapsto (Q_1, a\langle \mathsf{clos}(y, P_2)@\checkmark, \mathbf{0}\rangle b)$$

The applicable rule is now [S$\parr$], and so execution will context switch to P_2 after creating the session record for the new session with endpoints y and x:

$$(Q_1, a\langle \mathsf{clos}(y, P_2)@\checkmark, \mathbf{0}\rangle b) \Mapsto (P_2, y\langle \mathsf{nil}, Q_2\rangle x, a\langle \checkmark, \mathbf{0}\rangle b)$$

Process P_2 will execute as follows (rules [S$\mathbf{1}$],[S$\bot$] and [S$\bot$]):

$$(P_2, y\langle \mathsf{nil}, Q_2\rangle x, a\langle \checkmark, \mathbf{0}\rangle b) \Mapsto (Q_2, y\langle \checkmark, \mathbf{0}\rangle x, a\langle \checkmark, \mathbf{0}\rangle b) \Mapsto (Q_3, a\langle \checkmark, \mathbf{0}\rangle b) \Mapsto (\mathbf{0}, \emptyset)$$

consuming the appropriate ✓ and deallocating 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 $\mathsf{wait}\ b; \mathsf{wait}\ x; \mathbf{0}$, 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 execution of a $\mathsf{fwd}\ x\ y$ 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):

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

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

If we consider the execution of P (rules [SCut], [SCut], [S$\otimes$]):

$$(P, \emptyset) \Mapsto (\mathsf{cut}\ \{Q_1\ |a|\ \{a/d\}R_1\}, c\langle \mathsf{nil}, P_1\rangle b) \Mapsto (Q_1, a\langle \mathsf{nil}, R_1\rangle d, c\langle \mathsf{nil}, P_1\rangle b) \Mapsto (R_1, a\langle 1, Q_2\rangle d, c\langle \mathsf{nil}, P_1\rangle b)$$

The first three steps of the execution of P allocate the two session records and perform the write by Q_1. After firing rule [S$\otimes$], since the continuation type is negative, a context switch to R_1 and rule [S$\parr$] applies:

$$(R_1, a\langle 1, Q_2\rangle d, c\langle \mathsf{nil}, P_1\rangle b) \Mapsto (R_2, d\langle \mathsf{nil}, Q_2\rangle a, c\langle \mathsf{nil}, P_1\rangle b)$$

note how after R_1 performs its read, the read-to-write adjust operation swaps the endpoints of the session record, enabling the send by R_2 to be performed via rule [S$\otimes$]:

$$(R_2, d\langle \mathsf{nil}, Q_2\rangle a, c\langle \mathsf{nil}, P_1\rangle b) \Mapsto (Q_2, d\langle 2, R_3\rangle a, c\langle \mathsf{nil}, P_1\rangle b)$$

Rule [S$\otimes$] applies again, performing the write on c:

$$(Q_2, d\langle 2, R_3\rangle a, c\langle \mathsf{nil}, P_1\rangle b) \Mapsto (Q_3, d\langle 2, R_3\rangle a, c\langle 3, P_1\rangle b)$$

where Q_3 is a forwarder for endpoints a and c. Note that a is a read endpoint and c a write endpoint, as needed. Thus we apply rule [Sfwd], merging the two session records:

$$(Q_3, d\langle 2, R_3\rangle a, c\langle 3, P_1\rangle b) \Mapsto (P_1, d\langle 3@2, R_3\rangle b)$$

The execution can then proceed (rules [S$\parr$], [S$\parr$], [S$\mathbf{1}$], [S$\bot$]):

$$(P_1, d\langle 3@2, R_3\rangle b) \Mapsto (P_2, b\langle 2, R_3\rangle d) \Mapsto (P_3, b\langle \mathsf{nil}, R_3\rangle d) \Mapsto (R_3, b\langle \checkmark, R_3\rangle d) \Mapsto (\mathbf{0}, \emptyset)$$

Note the correct ordering in which the sent values are dequeued, where 3 is read before 2, as intended.

At this point, the reader may wonder about the correctness of the SAM's evaluation strategy as just discussed. 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 on the same session 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

To prove that the SAM adequately implements the operational semantics defined by reduction in CLL from Section 2, we need to show there are two way simulations between the two systems. However, there is a substantial gap between the language CLL, presented in an abstract algebraic style with its operational semantics defined by non-deterministic equational and rewriting systems, and an abstract machine such as the SAM, that evolves deterministically and represents the computation state by manipulating several "low-level" structures (heap, queues, etc). Even if the core SAM structure and transition rules are fairly simple, proving its correctness in relation to CLL is technically challenging, as is often the case with corresponding results relating higher-level languages and automata-style machines.

Therefore, to smoothly approach our results, we rely on a progressive build up. To construct the operational correspondence between the SAM execution and reduction in CLL, we first introduce an intermediate logical language CLLB that bridges between CLL and the SAM, conservatively extending CLL with a buffered cut construct, and approximates the effect of session queues in the SAM in a precise sense. The CLLB buffered cut has the form

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

and mediates all interactions between processes P and Q via a message queue q with two polarised endpoints a and b, where a of type A is held by process P and b of type B is held by process Q. The queue q stores values expressing messages communicated between channel endpoints. A polarised endpoint has the form x or $\overline{x}$. The endpoint marked $\overline{x}$ is the one allowing writes, the unmarked x is the one allowing reads, with exactly one of the two endpoints being marked at each moment. In a buffered cut the endpoint types A, B are related but do not need to be exact duals. In particular, the (session) type of the writer endpoint may be advanced "in time" with relation to the (session) type of the reader endpoint. This situation reflects that messages already sent by the writer process have been enqueued but have not yet been consumed by the reader. If the queue is empty, we must have $A = \overline{B}$. Thus a buffered cut with an empty queue corresponds exactly to a basic cut of CLL. Structural congruence for B (noted $\equiv^{\mathsf{B}}$) is obtained by extending $\equiv$ with commutative conversions for the buffered cut, listed in Fig. 8.

Definition 4.1 (Queue Values). Queue values and queues are defined by

$$\begin{array}{llllllllll}
V & ::= & \checkmark & \text{(Close token)} & \mid & \mathsf{step} & \text{(Recursion Unfold)} & q & ::= & \mathsf{nil} \mid V \mid q@q \quad \text{(Queue)} \\
 & \mid & \#\mathsf{l} & \text{(Selection Label)} & \mid & \mathsf{clos!}(x, P) & \text{(Exponential Closure)} & & & \\
 & \mid & \mathsf{clos}(x, P) & \text{(Linear Closure)} & \mid & \mathsf{ty}(T) & \text{(Type)} & & &
\end{array}$$

$$\begin{array}{lll}
\mathsf{cut}\ \{Q\ |a : A[q]b : B|\ P\}\ \equiv^{\mathsf{B}} \mathsf{cut}\ \{Q\ |b : B[q]a : A|\ P\} & [\mathrm{Bcomm}] & \\
\mathsf{cut}\ \{P\ |x[q]y|\ (Q \mid\mid R)\} \equiv^{\mathsf{B}} (\mathsf{cut}\ \{P\ |x[q]y|\ Q\}) \mid\mid R & [\mathrm{BM}] & y \in \mathsf{fn}(Q) \\
\mathsf{cut}\ \{P\ |x[q]z|\ (\mathsf{cut}\ \{Q\ |y[p]w|\ R\})\} \equiv^{\mathsf{B}} \mathsf{cut}\ \{Q\ |y[p]w|\ (\mathsf{cut}\ \{P\ |x[q]z|\ R\})\} & [\mathrm{BB}] & w, z \in \mathsf{fn}(R) \\
\mathsf{cut!}\ \{y.P\ |!x|\ (\mathsf{cut}\ \{Q\ |z[q]w|\ R\})\} \equiv^{\mathsf{B}} \mathsf{cut!}\ \{y.P\ |!x|\ \mathsf{cut}\ \{(y.P\ |!x|\ Q)\ |z[q]w|\ R\})\} & [\mathrm{BdistC!}] &
\end{array}$$

Fig. 8. Additional Structural Congruence Rules for CLLB.

$$\frac{P \vdash^{\mathsf{B}} \Delta', x : A; \Gamma \quad Q \vdash^{\mathsf{B}} \Delta, y : \overline{A}; \Gamma \quad A+}{\mathsf{cut}\ \{P\ |\overline{x} : A\ [\mathsf{nil}]\ y : \overline{A}|\ Q\}\ \vdash^{\mathsf{B}} \Delta', \Delta; \Gamma}\,[\mathrm{TcutE}] \qquad \frac{\mathsf{cut}\ \{\mathsf{close}\ x\ |\overline{x} : \mathbf{1}\ [q]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}{\mathsf{cut}\ \{0\ |\overline{x} : \emptyset[q@\checkmark]y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}\,[\mathrm{Tcut}\mathbf{1}]$$

$$\frac{\mathsf{cut}\ \{\mathsf{send}\ x(y.R); P\ |\overline{x} : T \otimes A\ [q]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}{\mathsf{cut}\ \{P\ |\overline{x} : A\ [q@\mathsf{clos}(y, R)]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}\,[\mathrm{Tcut}\otimes] \qquad \frac{\mathsf{cut}\ \{\#\mathsf{l}\ x; P\ |\overline{x} : \oplus_{\ell \in L} A_\ell\ [q]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}{\mathsf{cut}\ \{P\ |\overline{x} : A_{\#\mathsf{l}}\ [q@\#\mathsf{l}]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}\,[\mathrm{Tcut}\oplus]$$

$$\frac{\mathsf{cut}\ \{\mathsf{sendty}\ x(T); P\ |\overline{x} : \exists X.A\ [q]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}{\mathsf{cut}\ \{P\ |\overline{x} : \{T/X\}A\ [q@\mathsf{ty}(T)]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}\,[\mathrm{Tcut}\exists] \qquad \frac{\mathsf{cut}\ \{!x(z); P\ |\overline{x} :!A\ [q]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}{\mathsf{cut}\ \{0\ |\overline{x} : \emptyset\ [q@\mathsf{clos!}(z, P)]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}\,[\mathrm{Tcut!}]$$

$$\frac{\mathsf{cut}\ \{\mathsf{unfold}_\mu\ x; P\ |\overline{x} : \mu X.A\ [q]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}{\mathsf{cut}\ \{P\ |\overline{x} : \{\mu X.A/X\}A\ [q@\mathsf{step}]\ y : B|\ Q\}\ \vdash^{\mathsf{B}} \Delta; \Gamma}\,[\mathrm{Tcut}\mu]$$

Fig. 9. Additional typing rules for CLLB.

The value ✓ denotes session closure request, and #l a selection label issued from a selection process to an offer process. The linear closure $\mathsf{clos}(x, P)$ stores a suspended linear process P that interacts on fresh name x, such value is issued by a sender process to a receiver. The exponential closure $\mathsf{clos!}(x, P)$ is like process closure but for a replicable process. We also have $\mathsf{ty}(T)$, representing a type passed in communications via type send / receive operations, and the "step token" step representing a recursion unfold request. We use @ to denote (associative) concatenation operation of queues, with unit nil. Enqueue and dequeue operations occur respectively on the queue rhs and lhs.

The type system CLLB is obtained from CLL by replacing the [Tcut] rule with the five typing [Tcut-*] rules in Fig. 9. Apart from [TcutE], each [Tcut-*] rule applies to a distinguished positive type. We assume that for each of these rules the symmetrical one is defined.

We distinguish the type judgements as $P \vdash \Delta; \Gamma$ (or $P \vdash_{\mathsf{CLL}} \Delta; \Gamma$) for CLL and $P \vdash^{\mathsf{B}} \Delta; \Gamma$ for CLLB. The [TCutE] rule sets the endpoint's mode based on the cut type polarity, applicable whenever the queue is empty. The remaining rules relate queue contents with their corresponding (positive action) processes. Rule [Tcut⊗] 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⊕] and [Tcut!] apply a similar principle to the other possible queue contents. In [Tcut-**1**] and [Tcut!] the write endpoint is assigned the empty context $\emptyset$, to mark that the sending process has terminated (0), either by a close x action or by turning into a replicable value $!x(z); P$.

Reduction for CLLB (noted $\rightarrow^{\mathsf{B}}$) is obtained by replacing the CLL reduction → rules [fwd], [$\mathbf{1}\bot$], [$\otimes⅋$], [$\oplus\&$], [!?], [$\exists\forall$], [$\mu\nu$] and [corec], by the rules in Fig. 10. Essentially each principal cut reduction rule of CLL is replaced by a pair

$$
\begin{array}{ll}
\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\} & [\text{fwdp}] \\
\mathsf{cut}\ \{\mathsf{close}\ x\ |\overline{x}\ [q]\ y|\ Q\} \to^{\mathsf{B}} \mathsf{cut}\ \{0\ |\overline{x}\ [q@\checkmark]\ y|\ Q\} & [\mathbf{1}] \\
\mathsf{cut}\ \{0\ |\overline{x}\ [\checkmark]\ y|\ \mathsf{wait}\ y;P\} \to^{\mathsf{B}} P & [\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\} & [\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{r}} & [\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\} & [\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{r}} & [\&] \\
\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\} & [!] \\
\mathsf{cut}\ \{0\ |\overline{x}\ [\mathsf{clos!}(z,P)]\ y|\ ?y;Q\} \to^{\mathsf{B}} \mathsf{cut!}\ \{z.P\ |!y|\ Q\} & [?] \\
\mathsf{cut!}\ \{y.P\ |!x|\ \mathsf{call}\ x(z);Q\} \to \mathsf{cut!}\ \{y.P\ |!x|\ \mathsf{cut}\ \{P\ |y\ [\mathsf{nil}]\ z|\ Q\}^{\mathsf{p}}\} & [\text{call}] \\
\mathsf{cut}\ \{\mathsf{sendty}\ x(T);P\ |\overline{x}\ [q]\ y|\ R\} \to^{\mathsf{B}} \mathsf{cut}\ \{Q\ |\overline{x}\ [q@\mathsf{ty}(T)]\ y|\ R\} & [\exists] \\
\mathsf{cut}\ \{Q\ |\overline{x}\ [\mathsf{ty}(T)@q]\ y|\ \mathsf{recvty}\ y(X);R\} \to^{\mathsf{B}} \mathsf{cut}\ \{Q\ |\overline{x}\ [q]\ y|\ \{T/X\}R\}^{\mathsf{r}} & [\forall] \\
\mathsf{cut}\ \{\mathsf{unfold}_\mu\ x;P\ |\overline{x}\ [q]\ y|\ R\} \to^{\mathsf{B}} \mathsf{cut}\ \{P\ |\overline{x}\ [q@\mathsf{step}]\ y|\ R\} & [\mu] \\
\mathsf{cut}\ \{Q\ |\overline{x}\ [\mathsf{step}@q]\ y|\ \mathsf{rec}\ Y(u,\vec{w});Q\ [y,\vec{z}]\} \to^{\mathsf{B}} \mathsf{cut}\ \{Q\ |\overline{x}\ [q]\ y|\ \{y/u\}\{\vec{z}/\vec{w}\}\{(\mathsf{rec}\ Y(y,\vec{w});Q)/Y\}Q\}^{\mathsf{r}} & [\nu]
\end{array}
$$

Fig. 10. CLLB reduction $P \to^{\mathsf{B}} Q$.

of "positive" ($\to_p$) / "negative" ($\to_n$) 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 (rhs) of the queue and the rule for receive pops a session closure from the head (lhs) 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 the reduction rule [$\parr$] the polarities of the endpoints in the cuts occurring in the reductum depend on the types of the composed processes. To uniformly express the appropriate marking of endpoint polarities after reads we define the following convenient abbreviations:

Definition 4.2 (Adjusting polarities).

$$
\begin{array}{ll}
\mathsf{cut}\ \{Q\ |x : A[\mathsf{nil}]y : B|\ P\}^{\mathsf{p}} & \triangleq\ \textit{if}\, {+A}\ \textit{then}\ \mathsf{cut}\ \{Q\ |\overline{x} : A[\mathsf{nil}]y : B|\ P\}\ \textit{else}\ \mathsf{cut}\ \{Q\ |x : A[\mathsf{nil}]\overline{y} : B|\ P\} \\
\mathsf{cut}\ \{Q\ |\overline{x} : A[q]y : B|\ P\}^{\mathsf{r}} & \triangleq\ \textit{if}\ (q = \mathsf{nil})\ \textit{then}\ \mathsf{cut}\ \{Q\ |x : A[\mathsf{nil}]y : B|\ P\}^{\mathsf{p}}\ \textit{else}\ \mathsf{cut}\ \{Q\ |\overline{x} : A[q]y : B|\ P\}
\end{array}
$$

After a (negative) read operation, if the queue becomes empty, the type of the read endpoint may change polarity from negative to positive, and thus endpoints role must be swapped, for the cut to be typed (by [CutE]).

Definition 4.3 (Stable Process). We call a CLLB process stable if all its cuts have empty queues. Any CLL process S can be written as stable CLLB process $S^\dagger$, by replacing all its cuts by (empty) buffered cuts as follows:

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

Clearly $P \vdash \Delta;\Gamma$ implies $P^\dagger \vdash^{\mathsf{B}} \Delta;\Gamma$. Based on this correspondence, we may overload the notation $\mathsf{cut}\ \{R\ |x : A|\ Q\}$ to denote both a CLL cut or the corresponding empty-queued CLLB cut, if that makes sense in the context of use.

If Figure 11, we exemplify reduction of a CLLB process where processes S_1 and R_1 communicate on a single session. We could understand the system as the CLLB encoding $\mathsf{cut}\ \{S_1\ |\overline{x} : \mathbf{1} \otimes \mathbf{1} \otimes (\bot \parr \bot)[\mathsf{nil}]y : \bot \parr \bot \parr (\mathbf{1} \otimes \mathbf{1})|\ R_1\}$

$$\begin{array}{ll}
S_1 = \mathsf{send}\ x(c_1.\mathsf{close}\ c_1); S_2 & R_1 = \mathsf{recv}\ y(v_1); R_2 \\
S_2 = \mathsf{send}\ x(c_2.\mathsf{close}\ c_2); S_3 & R_2 = \mathsf{recv}\ y(v_2); R_3 \\
S_3 = \mathsf{recv}\ x(z); S_4 & R_3 = \mathsf{send}\ y(c_3.\mathsf{close}\ c_3); R_4 \\
S_4 = \mathsf{wait}\ z; \mathsf{wait}\ x; 0 & R_4 = \mathsf{wait}\ v1; \mathsf{wait}\ v2; \mathsf{close}\ y
\end{array}$$

$$\begin{array}{lr}
\mathsf{cut}\ \{S_1\ |\overline{x} : \mathbf{1}\otimes\mathbf{1}\otimes(\bot\parr\bot)[\mathsf{nil}]y : \bot\parr\bot\parr(\mathbf{1}\otimes\mathbf{1})|\ R_1\} \rightarrow^{\mathsf{B}} & [\otimes] \\
\mathsf{cut}\ \{S_2\ |\overline{x} : \mathbf{1}\otimes(\bot\parr\bot)[\mathsf{clos}(c_1.\mathsf{close}\ c_1)]y : \bot\parr\bot\parr(\mathbf{1}\otimes\mathbf{1})|\ R_1\} \rightarrow^{\mathsf{B}} & [\otimes] \\
\mathsf{cut}\ \{S_3\ |\overline{x} : \bot\parr\bot[\mathsf{clos}(c_1.\mathsf{close}\ c_1) :: \mathsf{clos}(c_2.\mathsf{close}\ c_2)]y : \bot\parr\bot\parr(\mathbf{1}\otimes\mathbf{1})|\ R_1\} \rightarrow^{\mathsf{B}} & [\parr] \\
\mathsf{cut}\ \{S_3\ |\overline{x} : \bot\parr\bot[\mathsf{clos}(c_2.\mathsf{close}\ c_2)]y : \bot\parr(\mathbf{1}\otimes\mathbf{1})|\ \mathsf{cut}\ \{\mathsf{close}\ c_1\ |\overline{c}_1 : \mathbf{1}[\mathsf{nil}]v_1 : \bot|\ R_2\}\} \equiv^{\mathsf{B}} & \\
\mathsf{cut}\ \{\mathsf{close}\ c_1\ |\overline{c}_1 : \mathbf{1}[\mathsf{nil}]v_1 : \bot|\ \mathsf{cut}\ \{S_3\ |\overline{x} : \bot\parr\bot[\mathsf{clos}(c_2.\mathsf{close}\ c_2)]y : \bot\parr(\mathbf{1}\otimes\mathbf{1})|\ R_2\}\} \rightarrow^{\mathsf{B}} & \\
\mathsf{cut}\ \{0\ |\overline{c}_1 : \emptyset[\checkmark]v_1 : \bot|\ \mathsf{cut}\ \{S_3\ |\overline{x} : \bot\parr\bot[\mathsf{clos}(c_2.\mathsf{close}\ c_2)]y : \bot\parr(\mathbf{1}\otimes\mathbf{1})|\ R_2\}\} \rightarrow^{\mathsf{B}} & \\
\mathsf{cut}\ \{\mathsf{close}\ c_2\ |\overline{c}_2 : \mathbf{1}[\mathsf{nil}]v_2 : \bot|\ \mathsf{cut}\ \{0\ |\overline{c}_1 : \emptyset[\checkmark]v_1 : \bot|\ \mathsf{cut}\ \{R_3\ |\overline{y} : \mathbf{1}\otimes\mathbf{1}[\mathsf{nil}]x : \bot\parr\bot|\ S_3\}\}\} \rightarrow^{\mathsf{B}} & \\
\mathsf{cut}\ \{0\ |\overline{c}_2 : \emptyset[\checkmark]v_2 : \bot|\ \mathsf{cut}\ \{0\ |\overline{c}_1 : \emptyset[\checkmark]v_1 : \bot|\ \mathsf{cut}\ \{R_3\ |\overline{y} : \mathbf{1}\otimes\mathbf{1}[\mathsf{nil}]x : \bot\parr\bot|\ S_3\}\}\} \rightarrow^{\mathsf{B}} & \\
\mathsf{cut}\ \{0\ |\overline{c}_2 : \emptyset[\checkmark]v_2 : \bot|\ \mathsf{cut}\ \{0\ |\overline{c}_1 : \emptyset[\checkmark]v_1 : \bot|\ \mathsf{cut}\ \{R_4\ |\overline{y} : \mathbf{1}[\mathsf{clos}(c_3.\mathsf{close}\ c_3)]x : \bot\parr\bot|\ S_3\}\}\} \equiv^{\mathsf{B}} & \\
\mathsf{cut}\ \{0\ |\overline{c}_2 : \emptyset[\checkmark]v_2 : \bot|\ \mathsf{cut}\ \{\mathsf{cut}\ \{0\ |\overline{c}_1 : \emptyset[\checkmark]v_1 : \bot|\ R_4\}\ |\overline{y} : \mathbf{1}[\mathsf{clos}(c_3.\mathsf{close}\ c_3)]x : \bot\parr\bot|\ S_3\}\} \rightarrow^{\mathsf{B}} & \\
\mathsf{cut}\ \{0\ |\overline{c}_2 : \emptyset[\checkmark]v_2 : \bot|\ \mathsf{cut}\ \{R_5\ |\overline{y} : \mathbf{1}[\mathsf{clos}(c_3.\mathsf{close}\ c_3)]x : \bot\parr\bot|\ S_3\}\} \equiv^{\mathsf{B}} & \\
\mathsf{cut}\ \{\mathsf{cut}\ \{0\ |\overline{c}_2 : \emptyset[\checkmark]v_2 : \bot|\ R_5\}\ |\overline{y} : \mathbf{1}[\mathsf{clos}(c_3.\mathsf{close}\ c_3)]x : \bot\parr\bot|\ S_3\} \rightarrow^{\mathsf{B}} & \\
\mathsf{cut}\ \{R_6\ |\overline{y} : \mathbf{1}[\mathsf{clos}(c_3.\mathsf{close}\ c_3)]x : \bot\parr\bot|\ S_3\} \rightarrow^{\mathsf{B}} & \\
\mathsf{cut}\ \{0\ |\overline{y} : \emptyset[\mathsf{clos}(c_3.\mathsf{close}\ c_3) :: \checkmark]x : \bot\parr\bot|\ S_3\} \rightarrow^{\mathsf{B}} & \\
\mathsf{cut}\ \{0\ |\overline{y} : \emptyset[\checkmark]x : \bot|\ \mathsf{cut}\ \{\mathsf{close}\ c_3\ |\overline{c}_3 : \mathbf{1}[\mathsf{nil}]z : \bot|\ S_4\}\} \rightarrow^{\mathsf{B}} & \\
\mathsf{cut}\ \{0\ |\overline{y} : \emptyset[\checkmark]x : \bot|\ \mathsf{cut}\ \{0\ |\overline{c}_3 : \emptyset[\checkmark]z : \bot|\ S_4\}\} \rightarrow^{\mathsf{B}} & \\
\mathsf{cut}\ \{0\ |\overline{y} : \emptyset[\checkmark]x : \bot|\ \mathsf{wait}\ x; 0\} \rightarrow^{\mathsf{B}} 0 &
\end{array}$$

$$\begin{array}{l}
\mathsf{cut}\ \{S_1\ |\overline{x} : \mathbf{1}\otimes\mathbf{1}\otimes(\bot\parr\bot)[\mathsf{nil}]y : \bot\parr\bot\parr(\mathbf{1}\otimes\mathbf{1})|\ R_1\} \rightarrow^{\mathsf{B}} \\
\mathsf{cut}\ \{S_2\ |\overline{x} : \mathbf{1}\otimes(\bot\parr\bot)[\mathsf{clos}(c_1.\mathsf{close}\ c_1)]y : \bot\parr\bot\parr(\mathbf{1}\otimes\mathbf{1})|\ R_1\} \rightarrow^{\mathsf{B}} \\
\mathsf{cut}\ \{S_2\ |\overline{x} : \mathbf{1}\otimes(\bot\parr\bot)[\mathsf{nil}]y : \bot\parr(\mathbf{1}\otimes\mathbf{1})|\ \mathsf{cut}\ \{\mathsf{close}\ c_1\ |\overline{c}_1 : \mathbf{1}[\mathsf{nil}]v_1 : \bot|\ R_2\}\} \rightarrow^{\mathsf{B}} \\
\mathsf{cut}\ \{S_3\ |\overline{x} : \bot\parr\bot[\mathsf{clos}(c_2.\mathsf{close}\ c_2)]y : \bot\parr(\mathbf{1}\otimes\mathbf{1})|\ \mathsf{cut}\ \{\mathsf{close}\ c_1\ |\overline{c}_1 : \mathbf{1}[\mathsf{nil}]v_1 : \bot|\ R_2\}\} \rightarrow^{\mathsf{B}} \cdots
\end{array}$$

Fig. 11. CLLB reduction (Examples)

of the CLL process $P = \mathsf{cut}\ \{S_1\ |x : \mathbf{1}\otimes\mathbf{1}\otimes(\bot\parr\bot)|\ R_1\{x/y\}\}$. We illustrate the flexibility introduced by the buffer mediated session interaction with an alternative ordering (bottom) of the first four reduction steps on the complete sequence (top). On top, two writes to the queue are anticipated, while below a the first send is immediately received by R_1 before the second send. In Section 4.2 below, we analyse the correspondence between CLL and CLLB reduction, where commutations such as the one just exemplified play an important role to show operational equivalences between the "synchronous" interactions of CLL and the apparently "asynchronous" buffer-mediated interactions of CLLB.

4.1 Preservation and Progress for CLLB

In this section, we prove the basic safety properties of CLLB: Preservation (Theorem 4.8) and Progress (Theorem 4.10). To reason about type derivations involving buffered cuts, we first formulate two convenient admissible inversion principles (Lemma 4.6 and Lemma 4.7). These principles, by monolithically aggregating applications of [TCut-$*$] rules of CLLB, allow us to talk in a uniform way about typing of values in queues and typing of processes connected by queues. To that end, we introduce an alternative typing systems for queues. Typing of queues q is specified by judgments the form $\Gamma; \Delta \vdash q : R \triangleright W$, where R and W are types. Intuitively, the judgment asserts that the queue q contains values typed in $\Gamma; \Delta$, sequenced in a consistent way for a process writing at type W and a receiver reading at type $\overline{R}$.

Definition 4.4 (Typing of Queues). Queues can be typed by the following admissible rules.

$$\Gamma; \vdash \mathsf{nil} : E \triangleright E \qquad \Gamma; \vdash \checkmark : \mathbf{1} \triangleright \emptyset \qquad \frac{P \vdash^{\mathsf{B}} \Delta_1, z : T; \Gamma \quad \Gamma; \Delta_2 \vdash q : E \triangleright F}{\Gamma; \Delta_1, \Delta_2 \vdash \mathsf{clos}(z, P)@q : T \otimes E \triangleright F} \qquad \frac{\Gamma; \Delta \vdash q : \{\mu X.E/X\}E \triangleright F}{\Gamma; \Delta \vdash \mathsf{step}@q : \mu X.E \triangleright F}$$

$$\frac{\Gamma; \Delta \vdash q : E_{\#\mathsf{l}} \triangleright F}{\Gamma; \Delta \vdash \#\mathsf{l}@q : \oplus_{\ell \in L} E_\ell \triangleright F} \qquad \frac{P \vdash^{\mathsf{B}} z : A; \Gamma}{\Gamma; \vdash \mathsf{clos!}(z, P) :!A \triangleright \emptyset} \qquad \frac{\Gamma; \Delta \vdash q : \{T/X\}E \triangleright F}{\Gamma; \Delta \vdash \mathsf{ty}(T)@q : \exists X.E \triangleright F}$$

Concatenation and splitting of queues preserves typing in the following sense.

Lemma 4.5. *Queue typings satisfy the following properties:*

(1) *(Interpolation) Let $\Gamma; \Delta \vdash q@q' : A \triangleright C$.*
Then there are B, Δ_1, Δ_2 such that $\Delta = \Delta_1, \Delta_2$, $\Gamma; \Delta_1 \vdash q : A \triangleright B$ and $\Gamma; \Delta_2 \vdash q' : B \triangleright C$.
(2) *(Transitivity) Let $\Gamma; \Delta_1 \vdash q : A \triangleright B$ and $\Gamma; \Delta_2 \vdash q' : B \triangleright C$. Then $\Gamma; \Delta_1, \Delta_2 \vdash q@q' : A \triangleright C$.*
(3) *(Transitivity) If B is negative and and $q : \overline{B} \triangleright C$ with C not positive then $q \neq \mathsf{nil}$.*

Proof. By induction on queue typing derivations. □

Lemma 4.6 (Non-full). *For $P \neq \mathsf{0}$ the following rule is (1) admissible and (2) invertible in* CLLB*:*

$$\frac{P \vdash^{\mathsf{B}} \Delta_P, x{:}A; \Gamma \quad Q \vdash^{\mathsf{B}} \Delta_Q, y{:}B; \Gamma \quad \Gamma; \Delta_q \vdash q : \overline{B} \triangleright A \quad B \textit{ negative}}{\mathsf{cut}\ \{P\ |\overline{x}{:}A\ [q]\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_q; \Gamma}$$

For inversion (2), the derivations for P and Q are sub-derivations of the conclusion.

Proof. See Appendix 10.1. By induction on the CLLB derivation of the conclusion. □

Notice that a CLL type, regarded as a session type, may terminate in either $\mathbf{1}$, $\perp$ or in a exponential $!A/?A$. We also have the following admissible inversion principle, which applies to full buffered cuts, that is, where the process P holding the writer endpoint has terminated execution, and the last value pushed into the queue is $\checkmark$ or $\mathsf{clos!}(z, R)$.

Lemma 4.7 (Full). *The following rule is (1) admissible and (2) invertible in* CLLB*:*

$$\frac{Q \vdash^{\mathsf{B}} \Delta_Q, y{:}B; \Gamma \quad \Gamma; \Delta_q \vdash q : \overline{B} \triangleright \emptyset \quad B \textit{ negative}}{\mathsf{cut}\ \{\mathsf{0}\ |\overline{x}{:}\emptyset\ [q]\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_Q, \Delta_q; \Gamma}$$

For (2), the derivation for Q is a sub-derivation of the conclusion. We also have $q = q'@\checkmark$ or $q = q'@\mathsf{clos!}(z, R)$ for some q'.

Proof. See Appendix 10.1. By induction on the derivation of the conclusion. □

Theorem 4.8 (Preservation). *Let $P \vdash^{\mathsf{B}} \Delta; \Gamma$. We have*

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

Proof. See Appendix 10.1. We verify that every conversion rule for $\equiv^{\mathsf{B}}$ and $\rightarrow^{\mathsf{B}}$ is well-typing preserving. □

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

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

Fig. 12. Observability Predicate $P\downarrow_x$.

To prove progress, we follow the technique of inductive observations introduced in [14], also adopted in [16, 68], that extends smoothly to the current setting of CLLB. A process P is *live* if and only if $P = C[Q]$, for some action process Q and some static process context C (a static process context is a process context where the hole lies only within the scope of static constructs mix or cuts, not behind an action prefix). We first show that any live process either reduces or offers an interaction. The observability predicate $P\downarrow_x$, defined in Fig. 12 (cf. [70]), characterises interactions of a process P with the environment on a free name x (notice that $P\downarrow_x$ implies $x \in \mathsf{fn}(P)$).

Lemma 4.9 (Liveness). *Let $P \vdash^{\mathsf{B}} \Delta; \Gamma$. If P is live then either $P\downarrow_x$ or $P \rightarrow^{\mathsf{B}}$.*

Proof. See Appendix 10.1. By induction on the derivation for $P \vdash \Delta; \Gamma$, and case analysis on the last typing rule. □

Theorem 4.10 (Progress). *Let $P \vdash \emptyset; \emptyset$ be a live process. Then, $P \rightarrow^{\mathsf{B}}$.*

Proof. Follows from Lemma 4.9, since $\Delta, \Gamma = \emptyset$ implies P has no free names. □

4.2 Correspondence between CLL and CLLB

In this section, we analyse the correspondence between CLL and CLLB, proving that the two languages simulate each other in a very tight sense. In one direction, the property directly follows from the form of CLLB reduction rules.

Lemma 4.11 (Simulation of CLL by CLLB). *Let $P \vdash \emptyset; \emptyset$. If $P \rightarrow Q$ then $P^{\dagger} \Rightarrow^{\mathsf{B}} Q^{\dagger}$.*

Proof. See Appendix 10.2. Each CLL reduction is simulated by two positive-negative CLLB reductions (e.g., CLLB $[\otimes⅋]$ by CLL $[\otimes]$ followed by $[⅋]$), and [fwd] by [fwdp]. □

On the opposite direction, the proof of the simulation is substantially more involved, since 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 introduce the following notations, which will allow us to express our operational correspondence results.

Definition 4.12 (Notation). We annotate CLLB reductions as follows:

1. Write $P \rightarrow^{\mathsf{B}p} Q$ for $P \rightarrow^{\mathsf{B}} Q$ if this reduction is positive (uses $[\mathbf{1}]$, $[\otimes]$, $[\oplus]$, $[!]$, $[\exists]$, or $[\mu]$).
2. Write $P \rightarrow^{\mathsf{B}n} Q$ for $P \rightarrow^{\mathsf{B}} Q$ if this reduction is negative or [call] (uses $[\bot]$, $[⅋]$, $[\&]$, $[?]$, [call], $[\forall]$, $[\nu]$ or $[\nu\mu]$).
3. Write $P \rightarrow^{\mathsf{B}a} Q$ for $P \rightarrow^{\mathsf{B}} Q$ if this reduction is by [fwdp].
4. Write $P \xrightarrow{\epsilon}{}^{\mathsf{B}a} Q$ for $P \rightarrow^{\mathsf{B}a} Q$ if this [fwdp] reduction acts on empty cuts.
5. Write $P \rightarrow^{\mathsf{Bap}} Q$ for $P \rightarrow^{\mathsf{B}} Q$ if this reduction is positive or a forwarder.
6. Write $P \rightarrow^{\mathsf{Br}} Q$ for a positive action on a buffered cut with empty queue immediately followed by a matching negative action on the very same cut (e.g., simulating an CLL reduction, cf. proof of Lemma 4.11).

Due to the progress property for CLLB (Theorem 4.10) and because queues are bounded by the size of positive/negative sections in types or recursion depth, after a sequence of positive or forwarder reductions a negative reduction consuming a queue value must occur. Theorem 4.15(2) states that every reduction sequence in CLLB, necessarily built of segments of the form $\Rightarrow^{\mathsf{Bap}} \rightarrow^{\mathsf{B}n}$, 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.

We now state and prove the mentioned results. They rely on several technical observations about CLLB reduction, which we collect in the following Lemmas. The first highlights some useful commutation properties.

Lemma 4.13 (Commutations). *The following commutation properties of reductions hold.*

(1) *Let $P_1 \rightarrow^{\mathsf{B}p} S \rightarrow^{\mathsf{B}n} P_2$. Then either (a) $P_1 \rightarrow^{\mathsf{Br}} P_2$, or (b) $P_1 \rightarrow^{\mathsf{B}n} S' \rightarrow^{\mathsf{B}p} P_2$ 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) *Let $P_1 \rightarrow^{\mathsf{Bap}} S \rightarrow^{\mathsf{B}n} P_2$. Then either (a) $P_1 \rightarrow^{\mathsf{Br}} P_2$, or (b) $P_1 \rightarrow^{\mathsf{B}n} S' \rightarrow^{\mathsf{Bap}} P_2$ for some S'.*
(4) *Let $P_1 \rightarrow^{\mathsf{Bap}} N \overset{\epsilon}{\Rightarrow}^{\mathsf{B}a} S \rightarrow^{\mathsf{Br}} P_2$. Then either (a) $P_1 \overset{\epsilon}{\rightarrow}^{\mathsf{B}a} N$ or (b) there is S' such that $P_1 \rightarrow^{\mathsf{Br}} S' \Rightarrow^{\mathsf{Bap}} P_2$.*

Proof. See Appendix 10.2. □

The postponing Lemma below makes precise the fact that all positive reductions and axioms may be postponed, except the ones matched by the next related negative reduction. In particular, this allows (2) a matching positive / negative pair or reductions to be anticipated and collapsed to an intial $\rightarrow^{\mathsf{Br}}$ reduction, and (1) the next negative reduction unmatched by an earlier positive reduction to be promoted to first reduction step.

Lemma 4.14 (Postponing). *Let $P \vdash^{\mathsf{B}} \emptyset;\emptyset$. If $P \Rightarrow^{\mathsf{Bap}} \rightarrow^{\mathsf{B}n} Q$ then either*

(1) *$P \rightarrow^{\mathsf{B}n} R$ and $R \Rightarrow^{\mathsf{Bap}} Q$ for some R, or;*
(2) *$P \overset{\epsilon}{\Rightarrow}^{\mathsf{B}a} \rightarrow^{\mathsf{Br}} R$ and $R \Rightarrow^{\mathsf{Bap}} Q$ for some R.*

Proof. See Appendix 10.2. The proof relies on the commutation properties of Lemma 4.13. □

Theorem 4.15 (Operational correspondence CLL-CLLB). *Let $P \vdash_{\mathsf{CLL}} \emptyset;\emptyset$.*

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

Proof. See Appendix 10.2. (1) Iterating Lemma 4.11. (2) Using Lemma 4.14. □

The results in this Section assert that CLL and CLLB essentially implement the same operational semantics, even given the "asynchronous flavour" of CLLB due to buffering. They will be leveraged in the next sections, to prove adequacy of the SAM execution, which relies on lower level data structure, w.r.t. CLL, whose reduction relation is formulated using algebraic rewriting of proofs.

5 THE LINEAR SAM AND ITS CORRECTNESS

In this section, we formally present the Linear SAM and prove its adequacy for executing CLL programs. More precisely we show that every execution trace of the SAM represents a correct process reduction sequence CLLB (and therefore of CLL, in light of Theorem 4.15). We first consider here the language without exponentials and mix, adopting a progressive development and presentation of our results. The basic multiplicative additive fragment of linear logic with recursion

$$
\begin{array}{lcll}
S & ::= & (P, H) & \text{Configuration} \\
R & ::= & x, y & \text{SRef} \\
H & ::= & (\mathit{SRef}, \mathit{SRef}) \longrightarrow \mathit{SessionRec} & \text{Heap} \\
R & ::= & x : A\langle q, P\rangle y : B & \text{Session Record} \\
q & ::= & \mathsf{nil} \mid V \mid V@q & \text{Queue} \\
\mathit{Val} & ::= & \checkmark & \text{Close Token} \\
 & \mid & \#\mathsf{l} & \text{Choice Label} \\
 & \mid & \mathsf{clos}(x, P) & \text{Process Closure} \\
 & \mid & \mathsf{ty}(T) & \text{Type Value} \\
 & \mid & \mathsf{step} & \text{Recursion Step}
\end{array}
$$

Fig. 13. The Linear SAM Components (no exponentials)

$$(\mathsf{cut}\ \{P\ |x : A|\ Q\}, H) \Rrightarrow (P, H[x : A\langle \mathsf{nil}, \{y/x\}Q\rangle y : \overline{A}])^{p} \qquad [\mathrm{SCut}]$$

$$(\mathsf{fwd}\ x\ y, H[z : A\langle q_1, Q\rangle x : \overline{B}][y : B\langle q_2, P\rangle w : C]) \Rrightarrow (P, H[z : A\langle q_2@q_1, Q\rangle w : C]) \qquad [\mathrm{Sfwd}]$$

$$(\mathsf{close}\ x, H[x : \mathbf{1}\langle q, P\rangle y : B]) \Rrightarrow (P, H[x : \emptyset\langle q@\checkmark, 0\rangle y : B]) \qquad [\mathrm{S}\mathbf{1}]$$

$$(\mathsf{wait}\ y; P, H[x : \emptyset\langle \checkmark, 0\rangle y : \bot]) \Rrightarrow (P, H) \qquad [\mathrm{S}\bot]$$

$$(\mathsf{send}\ x(z.R); Q, H[x : A \otimes C\langle q, P\rangle y : B]) \Rrightarrow (Q, H[x : C\langle q@\mathsf{clos}(z, R), P\rangle y : B])^{wr} \qquad [\mathrm{S}\otimes]$$

$$(\mathsf{recv}\ y(w : \overline{A}); Q, H[x{:}C\langle \mathsf{clos}(z : A, R)@q, P\rangle y{:}\overline{A} \parr D]) \Rrightarrow (Q, H[(x{:}C\langle q, P\rangle y{:}D)^{rw}][w{:}\overline{A}\langle \mathsf{nil}, R\rangle z{:}A])^{p} \qquad [\mathrm{S}\parr]$$

$$(\#\mathsf{l}\ x; Q, H[x : \oplus_{\ell \in L} A_\ell\langle q, P\rangle y : B]) \Rrightarrow (Q, H[x : A_{\#\mathsf{l}}\langle q@\#\mathsf{l}, P\rangle y : B])^{wr} \qquad [\mathrm{S}\oplus]$$

$$(\mathsf{case}\ y\ \{|\#\ell \in L{:}Q_\ell\}, H[x : A\langle \#\mathsf{l}@q, P\rangle y : \&_{\ell \in L} B_\ell]) \Rrightarrow (Q_{\#\mathsf{l}}, H[(x : A\langle q, P\rangle y : B_{\#\mathsf{l}})^{rw}]) \qquad [\mathrm{S}\&]$$

$$(\mathsf{sendty}\ x(T); Q, H[x : \exists X.A\langle q, P\rangle y : B]) \Rrightarrow (Q, H[x : \{T/X\}A\langle q@\mathsf{ty}(T), P\rangle y : B])^{wr} \qquad [\mathrm{S}\exists]$$

$$(\mathsf{recvty}\ y(X); Q, H[x : A\langle \mathsf{ty}(T)@q, P\rangle y : \forall X : B]) \Rrightarrow (\{T/X\}Q, H[(x : A\langle q, P\rangle y : \{T/X\}B)^{rw}]) \qquad [\mathrm{S}\forall]$$

$$(\mathsf{unfold}_\mu\ x; Q, H[x : \mu X.A\langle q, P\rangle y : B]) \Rrightarrow (P, H[x : \{\mu X.A/X\}A\langle q@\mathsf{step}, Q\rangle y : B]) \qquad [\mathrm{S}\mu]$$

$$\begin{array}{l}(\mathsf{rec}\ Y(u, \vec{w}); Q\ [y, \vec{z}], H[x : A\langle \mathsf{step}, P\rangle y : \nu X.B]) \Rrightarrow \\ \qquad\qquad (P, H[x : A\langle \mathsf{nil}, \{(\mathsf{rec}\ Y(u, \vec{w}); Q)/Y\}\{u, \vec{w}/y, \vec{z}\}Q\rangle y : \{\nu X.B/X\}B])^{p} \qquad [\mathrm{S}\nu]\end{array}$$

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

$$(P, H[x{:}A\langle q, Q\rangle y{:}B])^{wr} \triangleq \text{if } (A+) \text{ then } (P, H[x{:}A\langle q, Q\rangle y : B]) \text{ else } (Q, H[x{:}A\langle q, P\rangle y{:}B])$$

$$(P, H[x{:}A\langle \mathsf{nil}, Q\rangle y{:}B])^{p} \triangleq \text{if } (A+) \text{ then } (P, H[x{:}A\langle \mathsf{nil}, Q\rangle y{:}B]) \text{ else } (Q, H[y{:}B\langle \mathsf{nil}, P\rangle x{:}A])$$

Fig. 14. The Linear SAM Transition Rules

captures the essence of the main concepts behind the SAM design, and facilitates the presentation of results and proofs. The remaining constructs will be considered latter in the paper (Section 6).

5.1 Structure of the Linear SAM

The structure of the Linear SAM is presented in Figure 13. A configuration is a pair (P, H) where P is a (the currently active) CLL process and H a heap. A heap is a mutable store of session records of the form $x{:}A\langle q, P\rangle y{:}B$. In a session record $x{:}A\langle q, P\rangle y{:}B$, q is a queue storing the values already written, and P is a suspended CLL continuation process. We may establish an analogy between a session record and a frame in the call stack of a functional language, where q stores the "arguments" passed in a function call, and P represents the return "address". This analogy – which is of course just an approximation, since the SAM execution is more suggestive of co-routining rather than of a call-return protocol – will be further elaborated is Section 8, while discussing our proof-of-concept implementation.

The names x, y represent (unique) references or pointers to the record, x representing the write endpoint and y the read endpoint for the session. We could have modelled the heap with records accessed by a single reference z and distinguish endpoints roles using some kind of notational qualification, e.g. $z+$ and $z-$, but prefer to use this equivalent notation with distinct names x, y to facilitate the correspondence with CLLB). In any given heap H, all session endpoints are pairwise distinct and every session record is referred by its two unique endpoint references, as suggested by the heap representation as a map $(\mathit{SRef}, \mathit{SRef}) \rightarrow \mathit{SessionRec}$. The SAM operation does not rely on general type information, just on type polarity, which can be mostly statically determined. Nevertheless, in our formal description, we annotate session record endpoints with their types (A, B); this is convenient for clarifying the SAM behaviour and for the correctness proofs. We may omit type annotations when they may be easily recovered from the context.

The SAM queue values (cf. the CLLB queue values in Definition 4.1) are the close token ($\checkmark$), representing the session close handshake message, closures ($\mathsf{clos}(x, P)$), representing sessions passed in communications via send / receive operations, choice labels (#l), representing choices exercised in offer/choice operations, and types ($\mathsf{ty}(T)$), representing the types passed in communications via type send / receive operations. We also have the "step token" (step) representing a recursion unfold request. The SAM executes (co)-recursive sessions lazily, in the sense that positive sections of a session type are eagerly executed but only up to unfolds, and queues will therefore only store positive segments of outputs that fall within a single recursion body. More details about these and other aspects of how queue values are used during execution will be explained in detail shortly, while discussing the SAM transition rules. Notice that all processes stored in a SAM configurations (P, H), either as the active process P or the processes suspended in session records and closures in the heap H are source CLL processes, or, equivalently, stable CLLB processes with empty queues.

We now formally present the SAM execution rules in Figure 14. The SAM execution is driven by the top constructor of the process active in the current configuration – there is a single rule for each process construct. Any execution sequence is therefore *fully deterministic*. As already motivated in Section 3, at some well-defined steps the currently running process must yield execution to another process, suspended in a session record. Such context switching steps are absorbed by the transition rules, using the control operators on configurations presented in Figure 14 (bottom): namely init-adjust, write-to-read adjust, and read-to-write adjust; these control operators use type polarity information in a crucial way. We now discuss the transition rules in detail.

5.1.1 Cut and Forwarding. The [SCut] rule decomposes the cut, creates a fresh session record with the proper endpoints, and proceeds execution with the process that holds the writer endpoint (of positive type). As already discussed, even if the SAM operation is not generally guided by the structure of types, it relies on type polarity information. In the transition rules we thus define some auxiliary operations (Figure 14, bottom), to adjust control of execution based on type polarities. The operation $(P(x), H[x{:}A\langle \mathsf{nil}, Q(y)\rangle y{:}\overline{A}])^p$ (init adjust) is used to prepare a newly created session (a fresh session is always in write-mode, with empty queue), used in ([SCut], [S⅋], [Sν]). It essentially schedules process

$P(x)$ for execution if the type A of x is positive, which then becomes the write endpoint, and suspends $Q(y)$ at the negative endpoint y. If A is negative, the record is set conversely, with $Q(y)$ scheduled and $P(x)$ suspended.

The [SFwd] rule involves "merging" two session records into a single one. By typing, and the readiness property defined below (Definition 5.2), the session endpoints x and y must refer to different session records and be assigned dual types $\overline{B}$ and B. Without loss of generality (because of $\mathsf{fwd}\ x\ y \equiv \mathsf{fwd}\ y\ x$), we assume the forwarder $\mathsf{fwd}\ x\ y$ holds the read and write endpoints at respectively x and y, with $\overline{B}$ negative and B positive. The process $Q(z)$ suspended in the session record $z : A\langle q_1, Q\rangle x : \overline{B}$ has written (via z) the contents of q_1, whereas running process leading to $\mathsf{fwd}\ x\ y$ has previously written q_2. Thus, the process $P(w)$ suspended in the session record $y : B\langle q_2, P\rangle w : C$ is waiting to read $q_2@q_1$. Execution then continues with P, notice the two prior session records with endpoints z, x and y, w are replaced by a single session record with endpoints z, w.

5.1.2 *Send and Receive, Close and Wait.*

Rule [S$\mathbf{1}$] writes the close token to the queue, and switches control to the suspended process $P(y)$ holding the read endpoint. Rule [S$\bot$] reads the close token from the queue, disposes the session record, and continues execution.

Rule [S$\otimes$] writes a closure to the queue, and either continues (if the type of the continuation session type positive) or switches control to the suspended process $P(y)$, holding the read endpoint (if the continuation session type is negative). The control choice is implemented using the operation $(P, H[x{:}A\langle q, Q\rangle y{:}B])^{wr}$ (write-to-read adjust), which is used to prepare the state of the SAM after a positive (write) operation in an ongoing session ([S$\otimes$], [S$\oplus$], [S$\exists$]) – not needed for [S$\mathbf{1}$] since $\mathsf{close}\ x$ terminates the endpoint x usage. If the type of the write endpoint (positive polarity) x is (still) positive after the transition, execution continues with P and the session in write-mode. Otherwise, if the type of the write endpoint becomes negative, P has just enqueued the last value in the positive segment. The execution then context-switches: process P is suspended, and process Q is activated, to eventually read from the queue q at y, with the record in read-mode.

Rule [S$⅋$] reads a closure $\mathsf{clos}(z, R)$ from the queue and creates a fresh session record $(w{:}\overline{A}\langle \mathsf{nil}, R/Q\rangle z{:}A)$, to handle the interaction between R (at z) and the continuation Q of the receive process (at w). The scheduling choice of either R and Q with respect to their (implicit) cut is handled by init-adjust $(\ldots)^p$, that activates either R or Q depending on the polarity of the type of w (e.g., if $\overline{A}$ is positive, Q will write on w, so w is the positive endpoint and Q will be scheduled). A further endpoint adjustment may be required with respect to the ongoing read sequence on y, using read-to-write adjust (symmetric to write-to-read adjust). The operation $(x{:}A\langle q, Q\rangle y{:}B)^{rw}$ (read-to-write adjust) is used to readjust the polarity of session endpoints after a negative operation reads the last value from an ongoing session queue ([S$⅋$], [S$\&$], [S$\forall$]) – not needed for [S$\bot$] since $\mathsf{wait}\ y; P$ deallocates the terminated session. When the process holding the reader endpoint y empties the queue q, it becomes necessary to swap end-point polarities (the record switches to write mode, and y to positive polarity). Otherwise, the record endpoints retain the current polarities.

5.1.3 *Choice and Offer.*

The rules [S$\oplus$] and [S$\&$] follow the pattern of [S$\otimes$] and [S$⅋$], in a simpler setting. [S$\oplus$] writes a label to the queue, and [S$\&$] reads a label from the queue, and chooses the appropriate branch of the offer process. Write-to-read and read-to-write adjustments are applicable as expected.

5.1.4 *Type Send and Receive.*

In rule [S$\exists$] the active process writes a type value $\mathsf{ty}(T)$ to the appropriate queue, and in rule [S$\forall$] the active process reads a type from the queue, which is be substituted for the type parameter X in the body continuation code. In our description of the Full SAM in Section 6 we will use environments to track type bindings, rather than syntactical substitutions, which we adopt in this Section to avoid cluttering our presentation and proofs.

5.1.5 Unfold and Co-Recursion. The rules for unfold and co-recursion follow the rules for CLLB but sequentially schedule operations in a deterministic way that needs to escape the otherwise pervasive "write-bias" of the SAM. This is because recursive (inductive types) may introduce sequences of positive operations of unbounded length.

Recovering the basic example of the type for natural numbers as defined in Section 2.8

$$\mathsf{rec}\ \mathsf{Nat} = \oplus\{\,\#\mathsf{Z} : \mathbf{1}, \#\mathsf{S} : \mathsf{Nat}\,\}$$

we observe that the full representation of the natural N is a process of the form

$$\mathsf{N}(n : \mathsf{Nat}) = (\mathsf{unfold}_\mu\ n; \#\mathsf{S}\ n;)^{\mathsf{N}}\mathsf{unfold}_\mu\ n; \#\mathsf{Z}\ n; \mathsf{close}\ n$$

which exhibits a positive sequence with length of order N. To preserve the ability to statically bound queue lengths, the SAM always performs a context switching at unfold $\mathsf{unfold}_\mu\ n; P$ execution, yielding control to the dual co-recursive endpoint, that will start a read sequence until executing the matching co-recursor $\mathsf{rec}\ Y(, \vec{w}); Q\ [n, \vec{z}]$. More precisely, in rule [Tμ] process $\mathsf{unfold}_\mu\ n; Q$ writes the recursion unfold token step to the queue, and the SAM immediately switches context, yielding control to the process P that holds the dual endpoint, suspending the continuation Q in the session record. Rule [Tν], always executed when the session under focus is in read mode will read the recursion unfold token step, while the SAM switches context back to the code suspended after the triggering unfold, allowing subsequent writes to complete. Besides bounding queues, this strategy is consistent with the overall lazy evaluation strategy of the SAM, allowing inductive / co-inductive processes to interact via lazy streams, as generators and consumers. Our proofs of correctness show in detail how this execution strategy satisfies all the required SAM safety invariants.

We illustrate with the simple example in Figure 15. Here, program $\mathsf{main}()$ calls $\mathsf{printnat}(n)$ on $\mathsf{two}(n)$. Notice that although $\mathsf{two}(n)$ definitionally expands to the finite behavior

$$\mathsf{unfold}_\mu\ n; \#\mathsf{S}\ n; \mathsf{unfold}_\mu\ n; \#\mathsf{S}\ n; \mathsf{unfold}_\mu\ n; \#\mathsf{Z}\ n; \mathsf{close}\ n$$

the SAM execution consumes the session n lazily, with the recursive process coroutining (cf. a generator) with the co-recursive process, via the [Sμ] and [Sν] transitions. We conclude the present Section with the detailed proofs of correctness for the Linear SAM.

5.2 Correctness of the Linear SAM

We now state and prove that the Linear SAM correctly executes CLL programs. The main results state safety – every execution of the SAM corresponds to a reduction sequence of CLL processes – and progress – any SAM configuration reachable from a closed well-typed CLL process is either the terminated configuration $(\mathbf{0}, \emptyset)$, or still has a possible reduction. The proofs build (1) on the tight relationship between CLL and CLLB fully developed in Section 4.2, and (2) on the operational correspondences between reduction in CLLB and the SAM transition reduction. A key notion in this development is *configuration readiness* (Definition 5.2), which characterises the invariant properties of SAM states, which, together with basic typing constraints, are instrumental to prove the main safety and liveness properties.

In a ready configuration (P, H), whenever the running process P holds a session endpoint of negative type, and is therefore about to execute a negative action (e.g., a receive or offer action) on it, it will always find an appropriate value (resp. a closure or a label) to read from the appropriate session queue, an important requirement to establish progress. As a consequence, no busy waiting or context switching will be necessary for reading actions, since the sequential execution semantics of the SAM ensures that all values corresponding to a positive section of a session type have always been enqueued (written into the queue) by the "caller" process process before the "callee" process takes over (to

$$\begin{array}{l}
\mathsf{main}() = \mathsf{cut}\ \{\ \mathsf{two}(x)\ |x : \overline{Nat}|\ \mathsf{printnat}(x)\ \} \\
\mathsf{printnat}(n : \overline{\mathsf{Nat}}) = \mathsf{rec}\ Z(u);\mathsf{case}\ u\ \{\ \#\mathsf{Z} : \mathsf{wait}\ u; \mathbf{0},\ \ \#\mathsf{S} : Z(u)\ \}\ [n]
\end{array}$$

$$\begin{array}{l}
(\mathsf{main}(n), \emptyset) = \\
(\mathsf{cut}\ \{\ \mathsf{two}(n)\ |n : \overline{Nat}|\ \mathsf{printnat}(n)\ \}, \emptyset) \Mapsto \quad [\mathrm{SCut}] \\
(\mathsf{two}(x), [x\langle \mathsf{nil}, \mathsf{printnat}(y)\rangle y]) \Mapsto \quad [\mathrm{S}\mu] \\
(\mathsf{printnat}(y), [x\langle \mathsf{step}, \#\mathsf{S}\,x; \mathsf{one}(x)\rangle y]) \Mapsto \quad [\mathrm{S}\nu] \\
(\#\mathsf{S}\,x; \mathsf{one}(x), [x\langle \mathsf{nil}, \mathsf{case}\ y\ \{\ \#\mathsf{Z} : \mathsf{wait}\ y; \mathbf{0}, \#\mathsf{S} : \mathsf{printnat}(y)\ \}\rangle y]) \Mapsto \quad [\mathrm{S}\oplus] \\
(\mathsf{one}(x), [x\langle \#\mathsf{S}, \mathsf{case}\ y\ \{\ \#\mathsf{Z} : \mathsf{wait}\ y; \mathbf{0}, \#\mathsf{S} : \mathsf{printnat}(y)\ \}\rangle y]) \Mapsto \quad [\mathrm{S}\mu] \\
(\mathsf{case}\ y\ \{\ \#\mathsf{Z} : \mathsf{wait}\ y; \mathbf{0}, \#\mathsf{S} : \mathsf{printnat}(y)\ \}, [x\langle \#\mathsf{S}@\mathsf{step}, \mathsf{one}(x)\rangle y]) \Mapsto \quad [\mathrm{S}\&] \\
(\mathsf{printnat}(y), [x\langle \mathsf{step}, \mathsf{one}(x)\rangle y]) \Mapsto \quad [\mathrm{S}\nu] \\
(\#\mathsf{S}\,x; \mathsf{zero}(x), [x\langle \mathsf{nil}, \mathsf{case}\ y\ \{\ \#\mathsf{Z} : \mathsf{wait}\ y; \mathbf{0}, \#\mathsf{S} : \mathsf{printnat}(y)\ \}\rangle y]) \Mapsto \quad [\mathrm{S}\oplus] \\
(\mathsf{zero}(x), [x\langle \#\mathsf{S}, \mathsf{case}\ y\ \{\ \#\mathsf{Z} : \mathsf{wait}\ y; \mathbf{0}, \#\mathsf{S} : \mathsf{printnat}(y)\ \}\rangle y]) \Mapsto \quad [\mathrm{S}\mu] \\
(\mathsf{case}\ y\ \{\ \#\mathsf{Z} : \mathsf{wait}\ y; \mathbf{0}, \#\mathsf{S} : \mathsf{printnat}(y)\ \}, [x\langle \#\mathsf{S}@\mathsf{step}, \mathsf{zero}(x)\rangle y]) \Mapsto \quad [\mathrm{S}\&] \\
(\mathsf{printnat}(y), [x\langle \mathsf{step}, \mathsf{zero}(x)\rangle y]) \Mapsto \quad [\mathrm{S}\nu] \\
(\#\mathsf{Z}\,x; \mathsf{close}\ x, [x\langle \mathsf{nil}, \mathsf{case}\ y\ \{\ \#\mathsf{Z} : \mathsf{wait}\ y; \mathbf{0}, \#\mathsf{S} : \mathsf{printnat}(y)\ \}\rangle y]) \Mapsto \quad [\mathrm{S}\oplus] \\
(\mathsf{close}\ x, [x\langle \#\mathsf{Z}, \mathsf{case}\ y\ \{\ \#\mathsf{Z} : \mathsf{wait}\ y; \mathbf{0}, \#\mathsf{S} : \mathsf{printnat}(y)\ \}\rangle y]) \Mapsto \quad [\mathrm{S}\mathbf{1}] \\
(\mathsf{case}\ y\ \{\ \#\mathsf{Z} : \mathsf{wait}\ y; \mathbf{0}, \#\mathsf{S} : \mathsf{printnat}(y)\ \}, [x\langle \#\mathsf{Z}@\checkmark, \mathbf{0}\rangle y]) \Mapsto \quad [\mathrm{S}\&] \\
(\mathsf{wait}\ y; \mathbf{0}, [x\langle \checkmark, \mathbf{0}\rangle y]) \Mapsto \quad [\mathrm{S}\bot] \\
(\mathbf{0}, \emptyset)
\end{array}$$

Fig. 15. Example Trace for Recursion in the SAM

read from the queue). As discussed in Section 3 it might not seem obvious whether all such input endpoint (including endpoints passed around in higher-order communications via send / receive) always refer to non-empty queues in the specific execution strategy fixed for the SAM.

The effect of such interleaved read and write phases are captured in the notion of ready configuration by requiring every session record $x{:}A\langle q, R\rangle y{:}B$ stored in the heap to necessarily be in one of two exclusive "modes": *write-mode* or *read-mode*. To intuitively motivate these ideas, we illustrate the life cycle of a session record (Figure 16). We structure our discussion by first covering the SAM behaviour on the simpler recursion-free fragment, and introduce recursion afterwards as a second step.

A session record $x{:}A\langle q, R\rangle y{:}B$ is in *write-mode* if its free access point is the write endpoint x, owned by some process P about to write to the queue q, so the type A is positive. In this case, $y \in \mathsf{fn}(R)$ and $R = R(y)$ is the process holding the other dual endpoint of the session, waiting for the writing phase to complete. A writing phase may terminate either because P closes the session ($P = \mathsf{close}\ x$ and A transitions to $\emptyset$ by [S$\mathbf{1}$]) or the write endpoint type A becomes negative (that is, the session type changes polarity from positive to negative). If A becomes non-positive after a write step, then the session record will switch to read mode (by write-to-read adjust), with process $R(y)$ starting to read from queue q at endpoint y (illustrated by the transition from write-mode to read-mode in Figure 16 (top, left to right)). Notice that we with represent the active endpoint by a hollow bullet ∘ and the suspended endpoint by a filled bullet •.

A session record $x{:}A\langle q, P\rangle y{:}B$ is in *read-mode* if its free access point is the read endpoint y, with reading being performed by some process $Q(y)$ owning endpoint y (so $y \notin \mathsf{fn}(P)$ by linearity). In this case, P is the suspended process that, having written to the queue, either closed the session ($P = \mathbf{0}$ and $A = \emptyset$) or terminated the write phase due to session polarity inversion and is now waiting to read on x (with $x \in \mathsf{fn}(P)$ and A negative). The queue q must be

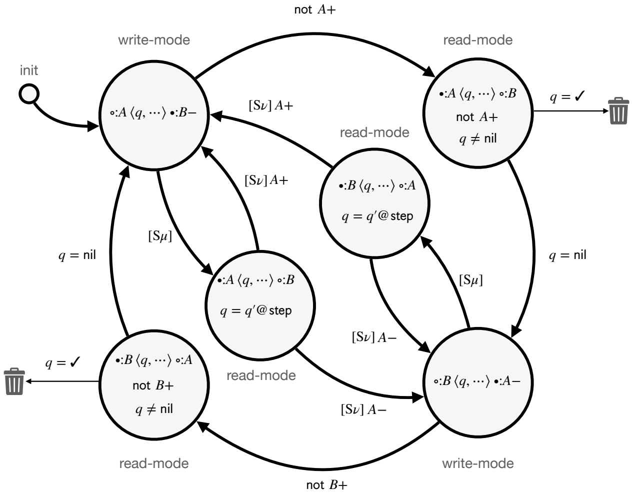
not $A+$
write-mode
read-mode
init
$\circ{:}A\,\langle q, \cdots\rangle\,\bullet{:}B-$
[Sν] $A+$
$\bullet{:}A\,\langle q, \cdots\rangle\,\circ{:}B$
not $A+$
$q \neq$ nil
$q = ✓$
read-mode
[Sν] $A+$
$\bullet{:}B\,\langle q, \cdots\rangle\,\circ{:}A$
$q = q'$@step
[Sμ]
$q =$ nil
$q =$ nil
[Sμ]
$\bullet{:}A\,\langle q, \cdots\rangle\,\circ{:}B$
$q = q'$@step
[Sν] $A-$
$q = ✓$
$\bullet{:}B\,\langle q, \cdots\rangle\,\circ{:}A$
not $B+$
$q \neq$ nil
read-mode
$\circ{:}B\,\langle q, \cdots\rangle\,\bullet{:}A-$
[Sν] $A-$
read-mode
write-mode
not $B+$

Fig. 16. Session Record Life Cycle.

non-empty ($q \neq$ nil). A reading phase terminates when the queue becomes empty after a read operations. If the last value read is ✓, the session record reaches the end of its life, the session terminating by [S⊥]. Otherwise, the read endpoint type B becomes positive (because if the queue is empty, it is the dual of A which is known to be negative). This is because the session on y by Q is now changing polarity from negative to positive. The session record will then switch write mode as an effect of read-to-write adjust, which involves swapping the endpoints and let Q proceed with writing on y. (illustrated by the transition from read-mode to write-mode in Figure 16 (right, top to bottom)).

In our session life cycle sketch, the evolution from bottom right (write-mode) to top left (write mode) via bottom left, is symmetric to the evolution just described, with endpoints reversing roles. This reflects the iterated general sequence of communications between two processes $P(x)$ and $Q(y)$ interacting in a session with endpoints x and y: write phase by $P(x)$; read phase by $Q(y)$; write phase by $Q(y)$; read phase by $P(x)$, and so on.

Having explained the SAM behaviour on the basic CLL fragment, we can now cover the recursion and co-recursion constructs, and the specific process co-routing pattern associated to it. Unfolding of recursion is lazily controlled by the passing of step tokens from the inductive typed endpoint to the co-inductive typed endpoint. This is illustrated in the figure by the transitions labeled with “unfold” and “corec”. When a session record $x{:}A\langle q, R\rangle y{:}B$ is in write mode but the current operation is unfold (so that $A = \mu X.C$ is a positive) the record switches to read mode by [Sμ] as $x{:}\{A/X\}C\langle q@\mathsf{step}, R\rangle y{:}B$. The record will continue in read mode until the matching corec becomes exposed, in a state like $x{:}\{A/X\}C\langle \mathsf{step}, R\rangle y{:}\nu X.\overline{C}$. It will then transition by [Sν] to write mode as $x{:}\{A/X\}C\langle \mathsf{nil}, R\rangle y{:}\{B/X\}\overline{C}$ or $y{:}\{B/X\}\overline{C}\langle \mathsf{nil}, R\rangle x{:}\{A/X\}C$, depending on which of $\{A/X\}C$ or $\{B/X\}\overline{C}$ is positive. Notice that this protocol ensures that queues may contain at most one step token at any given time, always in the last queue position, and only in read-mode session records. Recursion/co-recursion related phases are also depicted in Figure 16 in the two states in the center, connected by [Sμ] or [Sν] transitions.

Motivated by the prior analysis, we now present the following definitions, which characterise the key invariants of safe SAM configurations, based on session record modes and related properties. We say that a queue q is step-terminated if $q = q'@\mathsf{step}$ and $\mathsf{step} \notin q'$.

Definition 5.1 (Modes). In a well-moded session record $x{:}A\langle q, R\rangle y{:}B$, at most one of x, y may occur free in R. Moreover, well-moded session record is in:

(1) write-mode, if $y \in \mathsf{fn}(R)$, A is positive and $\mathsf{step} \notin q$.
(2) read-mode, if $x \in \mathsf{fn}(R)$ or $R = \mathbf{0}$ and either
  (a) A is not positive and $\mathsf{step} \notin q$, or
  (b) q is step-terminated.

Notice that read and write modes are exclusive, if a record is in write-mode then $y \in \mathsf{fn}(R)$, while a record in read-mode must have $x \in \mathsf{fn}(R)$ (would contradict typing of cut) or $R = \mathbf{0}$ (would contradict $y \in \mathsf{fn}(R)$). As a consequence of 2(b) and queue typing (cf. Lemma 4.5(3)), we have $q \neq \mathsf{nil}$ for any record in read-mode.

Definition 5.2 (Ready Configuration). A SAM configuration (P, H) is ready if any session record in H is either in write-mode or read-mode.

To prove our adequacy results, we need to formally relate SAM transitions with reductions in CLL. As explained before Section 4, we use CLLB as a bridge. To that end, we define the following encoding of CLLB processes, that satisfy certain structural conditions, into SAM states.

Definition 5.3 (Encoding CLLB *to SAM).* For well-typed CLLB processes P, P' and well-formed heaps H, H' let relation $(P, H) \overset{\mathsf{enc}}{\Mapsto} (P', H')$ be defined by the rules.

$$(\mathsf{cut}\ \{P(x)\ |\overline{x}{:}A\ [q]\ y{:}B|\ Q(y)\}, H) \overset{\mathsf{enc}}{\Mapsto} (P(x), H[x{:}A\langle q, Q(y)\rangle y{:}B]) \qquad [\text{Cut-write}] \qquad (\ x\,\langle -\rangle\, y \text{ in write mode })$$

$$(\mathsf{cut}\ \{P(x)\ |\overline{x}{:}A\ [q]\ y{:}B]|\ Q(y)\}, H) \overset{\mathsf{enc}}{\Mapsto} (Q(y), H[x{:}A\langle q, P(x)\rangle y{:}B]) \qquad [\text{Cut-read}] \qquad (\ x\,\langle -\rangle\, y \text{ in read mode })$$

Given $P \vdash^{\mathsf{B}} \emptyset$, we write $P \overset{\mathsf{enc*}}{\Mapsto} C$ for $(P, \emptyset) \overset{\mathsf{enc*}}{\Mapsto} C$ and $enc(P) = C$ for $P \overset{\mathsf{enc*}}{\Mapsto} (A, H)$, for $\Delta \vdash_{\mathsf{CLL}} A$ with $A = \mathcal{A}$ or $A = \mathbf{0}$.

Given a well-typed CLLB process P, whenever $enc(P)$ is defined, it gives a SAM configuration $(\mathcal{A}, H)$ where all top-level cuts in P covering $\mathcal{A}$ are represented in heap H by session records, leaving the unguarded action $\mathcal{A}$ as the active process in the configuration. The well-formedness conditions in the definition of $enc(.)$ ensure that all generated session records are always well-moded, and that SAM configurations are always well-formed, in particular that all processes embedded in (A, H), be it in the active position P or suspended in heap stored session records or closures are always "source level" CLL processes (cf. CLLB processes with empty queues).

We write $C \overset{\mathsf{cut*}}{\Mapsto} \mathcal{D}$ if $C \overset{*}{\Mapsto} \mathcal{D}$ by repeated use of SAM cut transitions [SCut]. We have

Lemma 5.4 (Readiness of Encoding). *Basic properties of* $\overset{\mathsf{enc}}{\Mapsto}$.

(1) *If C is ready and $C \overset{\mathsf{enc}}{\Mapsto} \mathcal{D}$, then $C \Mapsto \mathcal{D}$ with $\mathcal{D}$ is ready.*
(2) *If (P, H) is ready and $P \vdash_{\mathsf{CLL}} \Delta$ then $(P, H) \overset{\mathsf{enc*}}{\Mapsto} (A, H') = C$ and $(P, H) \overset{\mathsf{cut*}}{\Mapsto} C$, for $A = \mathcal{A}$ or $A = \mathbf{0}$.*
(3) *If $P \vdash_{\mathsf{CLL}} \emptyset$ then $enc(P)$ is ready.*

Proof. (1) Any $\overset{\mathsf{enc}}{\Mapsto}$ step in a ready configuration creates a new session record in either write- or read-mode. (2) For a CLL process P, $(P, H) \overset{\mathsf{enc}}{\Mapsto} C$ must be by [Cut-write] coinciding with [SCut]. (3) By (1,2) applied to $(P, \emptyset) \overset{\mathsf{enc*}}{\Mapsto} enc(P)$. □

A SAM configuration C is live if $C = (\mathcal{A}, H)$.

Lemma 5.5 (SAM-CLLB Step Safety). *Let $P \vdash^{\mathsf{B}} \emptyset$ and $enc(P)$ a ready SAM configuration. If $enc(P)$ is live then (1) there is C ready such that $enc(P) \Mapsto C$ and (2) there is Q such that $P \to^{\mathsf{B}} Q$ and $Q \overset{\mathsf{enc*}}{\Mapsto} C \overset{\mathsf{cut*}}{\Mapsto} enc(Q)$.*

Proof. See Appendix 10.3. We show that each SAM transition corresponds to a reduction in the CLLB process encoded by the machine configuration. An auxiliary technical result states for every ready configuration $enc(P) = (\mathcal{A}, H)$ there are CLLB process contexts E, F such that $P \equiv E[F[\mathcal{A}]]$. We then show that for every SAM transition $(\mathcal{A}, H) \Mapsto C$ $F[\mathcal{A}]$ is a principal cut redex such that $F[\mathcal{A}] \to^{\mathsf{B}} R$ and thus $P = E[F[\mathcal{A}]] \to^{\mathsf{B}} E[R] \overset{\mathsf{enc}*}{\Mapsto} C$. □

We then have

Theorem 5.6 (Soundness wrt CLLB). *Let $P \vdash^{\mathsf{B}} \emptyset$ and $enc(P)$ a ready SAM configuration. If $enc(P) \overset{*}{\Mapsto} C$ then there is Q such that $P \Rightarrow^{\mathsf{B}} Q$ and $Q \overset{\mathsf{enc}*}{\Mapsto} C$.*

Proof. In Appendix 10.3. By induction on the number n of transition steps in $enc(P) \overset{*}{\Mapsto} C$. □

Combining the previous results with the operational correspondence between CLL and CLLB (Theorem 4.15) we obtain an overall soundness result for the Linear SAM relative to CLL. Every SAM execution starting from a well-typed closed CLL process P up to a a configuration C corresponds to a CLL reduction sequence that starts in P and terminates in a CLL process Q represented by C up to some anticipated positive (enqueueing) and forwarding operations.

Theorem 5.7 (Soundness wrt CLL). *Let $P \vdash_{\mathsf{CLL}} \emptyset$.*
If $(P, \emptyset) \overset{*}{\Mapsto} C$ then there is Q such that $P \Rightarrow_{\mathsf{CLL}} Q$ and $Q^{\dagger} \Rightarrow^{\mathsf{Bap}} \overset{\mathsf{enc}*}{\Mapsto} C$.

Proof. Let $enc(P) \overset{*}{\Mapsto} C$. By Lemma 5.4 (2), $enc(P)$ is ready. By Theorem 5.6, there is Q' such that $P \Rightarrow^{\mathsf{B}} Q'$ and $Q' \overset{\mathsf{enc}*}{\Mapsto} C$. By Theorem 4.15 (2), there is Q such that $P \Rightarrow_{\mathsf{CLL}} Q$ and $Q^{\dagger} \Rightarrow^{\mathsf{Bap}} Q'$. We conclude $Q^{\dagger} \Rightarrow^{\mathsf{Bap}} Q' \overset{\mathsf{enc}*}{\Mapsto} C$. □

We now state our progress result. Any SAM execution starting from a well-typed process either reaches the inaction process in the empty heap, with all store deallocated, or is able to make a further transition.

Theorem 5.8 (Progress). *Let $P \vdash_{\mathsf{CLL}} \emptyset$. If $(P, \emptyset) \overset{*}{\Mapsto} C$ then either $C = (0, \emptyset)$ or there is C' such that $C \Mapsto C'$.*

Proof. We first prove the following property (P 5.8): Let $P \vdash_{\mathsf{CLLB}} \emptyset$, and $P \overset{\mathsf{enc}*}{\Mapsto} \mathcal{D} \overset{\mathsf{cut}*}{\Mapsto} C \overset{\mathsf{cut}*}{\Mapsto} enc(P)$, then either $C = (0, \emptyset)$ or there is C' such that $C \Mapsto C'$. (Proof: Either $C \overset{\mathsf{cut}}{\Mapsto} C'$ or $enq(P) = C = (A, H)$ with $A = 0$ or $A = \mathcal{A}$. If $A = 0$, we must have $H = \emptyset$. Otherwise C is live, and by Lemma 5.5 there is C' such that $C \Mapsto C'$).

To prove the Theorem, we break the hyp. $(P, \emptyset) \overset{*}{\Mapsto} C$ (by determinism of $\Mapsto$) in the two cases: either (a) $(P, \emptyset) \overset{\mathsf{cut}*}{\Mapsto} C \overset{\mathsf{cut}*}{\Mapsto} enc(P)$ or (b) $(P, \emptyset) \overset{\mathsf{cut}*}{\Mapsto} enc(P) \overset{*}{\Mapsto} C$. For (a), we conclude by P 5.8. For (b), we proceed by induction on the number n of transition steps in $enc(P) \overset{n}{\Mapsto} C$. (Base case $n = 0$) Then $enc(P) = C$ and we conclude by P 5.8. (Inductive case $n = 1 + n'$) Then $enc(P) \Mapsto \mathcal{D} \overset{n'}{\Mapsto} C$, so $enc(P)$ is live. By Lemma 5.5, there is Q such that $P \to^{\mathsf{B}} Q$ and $Q \overset{\mathsf{enc}*}{\Mapsto} \mathcal{D} \overset{\mathsf{cut}*}{\Mapsto} enc(Q)$. We consider two cases, either (1) $\mathcal{D} \overset{\mathsf{cut}*}{\Mapsto} enc(Q) \overset{n''}{\Mapsto} C$ with $n'' < n$ or (2) $\mathcal{D} \overset{\mathsf{cut}*}{\Mapsto} C \overset{\mathsf{cut}*}{\Mapsto} enc(Q)$. For (1) we conclude by the i.h.. For (2), we conclude by P 5.8. □

Theorem 5.8 asserts that SAM executions on well-typed processes never get stuck, but also a "no garbage left" property, in the sense that a terminated computation always leaves an empty heap. Although termination is not a consequence of our results just presented, it is a consequence of the soundness property (Theorem 5.7) via the strong normalisation property of CLL [65, 68].

$$
\begin{array}{llll}
S & ::= & (\mathcal{E}, P, H) & \text{State} \\
R & ::= & a, b & \text{SRef} \\
H & ::= & (\mathit{SRef}, \mathit{SRef}) \longrightarrow \mathit{SessionRec} & \text{Heap} \\
\mathit{SessionRec} & ::= & a\langle q, \mathcal{E}, P\rangle b & \\
q & ::= & \mathsf{nil} \mid \mathit{Val} \mid q@q & \text{Queue} \\
\mathit{Val} & ::= & \checkmark & \text{Close token} \\
 & \mid & \#\mathsf{l} & \text{Choice label} \\
 & \mid & \mathsf{clos}(z, \mathcal{E}, P) & \text{Linear Closure} \\
 & \mid & \mathsf{clos!}(z, \mathcal{E}, P) & \text{Exponential Closure} \\
 & \mid & \mathsf{ty}(T) & \text{Type Value} \\
 & \mid & \mathsf{step} & \text{Recursion Step} \\
\mathcal{E}, \mathcal{G}, \mathcal{F} & ::= & \mathit{Name} \longrightarrow (\mathit{SRef} \cup \mathsf{clos!}(z, \mathcal{E}, P)) & \text{Environment}
\end{array}
$$

Fig. 17. The Environment based SAM for full CLL.

6 THE LINEAR SAM FOR FULL CLL

In the previous sections, to simplify the formal presentation of the SAM, the proofs of its meta-theory, and discuss its design at a more intuitive level, we adopted an abstract formalisation of the operational semantics, relying on (implicit) α-conversion, and overloading language syntax names for the intended heap references for session records. However, to handle our full language with exponentials and bring the SAM execution model closer to a low level implementation, it now introduce in it a more traditional environment structure allowing us to get rid of syntactic substitutions from the definition of transition rules, as implemented in most abstract machines since Landin's SECD [48]. Notice that, to avoid excessive notational burden, we do not use the environment to track type substitutions (although we might have done so), so all types in configurations are closed. Another reason for doing so is that our dynamical use of types in the SAM is mostly computationally irrelevant, and essentially useful for our correctness proofs.

We thus reformulate the SAM as presented in Figure 17. Formally, a SAM configuration is a triple $(\mathcal{E}, P, H)$ where $\mathcal{E}$ is an environment. Formally, an environment $\mathcal{G}$ is a finite map that sends (linear) names to heap record endpoint *SRef* and exponential names to replicated process closures, injective on linear names. These heap references are freshly allocated and unique, thus avoiding any clashes and enforcing proper static scoping. Process 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.

In Figure 18 we present the execution rules for the environment-based SAM. All rules except those for exponentials have already been essentially presented in Fig. 14 and discussed in previous sections. The only changes from are due to the presence of environments, which record the bindings for free names in the code.

Recall that the SAM operation relies on some type information, namely, polarity information is required in the cut rule, in the rules for session receive [$\parr$] and replicated session invocation [Scall]. In the basic first-order typed language, such polarity information may be statically identified at type-checking and used to tag the code before execution. However, in the presence of type parametricity as we now consider, the polarity of the instantiation of a parametric type may depend in general on the polarity of the types used to substitute its free type variables, so, for simplicity, we assume that types are explicitly inspected at runtime for polarity (see, e.g., in rule [SCut]).

$$(\mathcal{E}, \mathsf{cut}\ \{P \mid x : A \mid Q\}, H) \Mapsto (\mathcal{G}, P, H[a : A\langle \mathsf{nil}, \mathcal{F}, Q\rangle b{:}\overline{A}])^{p} \qquad [\text{SCut}]$$
$$a, b = \text{new}, \mathcal{G} = \mathcal{E}\{a/x\}, \mathcal{F} = \mathcal{E}\{b/x\}, A^{+}$$

$$(\mathcal{E}, \mathsf{fwd}\ x\ y, H[c : A\langle q_1, \mathcal{G}, Q\rangle a : \overline{B}][b : D\langle q_2, \mathcal{F}, P\rangle c : C]) \Mapsto (\mathcal{F}, P, H[c : A\langle q_2 @ q_1, \mathcal{G}, Q\rangle d : C]) \qquad [\text{Sfwd}]$$
$$a = \mathcal{E}(x), b = \mathcal{E}(y)$$

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

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

$$(\mathcal{E}, \mathsf{send}\ x(z.R); Q, H[a{:}A \otimes C\langle q, \mathcal{G}, P\rangle b{:}B]) \Mapsto (\mathcal{E}, Q, H[a{:}C\langle q @ \mathsf{clos}(z{:}A, \mathcal{E}, R), \mathcal{G}, P\rangle b{:}B])^{wr} \qquad [\text{S}\otimes]$$
$$a = \mathcal{E}(x)$$

$$(\mathcal{E}, \mathsf{recv}\ y(w); Q, H[a{:}C\langle \mathsf{clos}(z{:}A, \mathcal{F}, R) @ q, \mathcal{G}, P\rangle b{:}\overline{A} \parr D]) \Mapsto$$
$$(\mathcal{E}', Q, H[(a{:}C\langle q, \mathcal{G}, P\rangle b{:}D)^{rw}][e{:}\overline{A}\langle \mathsf{nil}, \mathcal{F}', R\rangle f{:}A])^{p} \qquad [\text{S}\parr]$$
$$e, f = \text{new}, b = \mathcal{E}(y), \mathcal{E}' = \mathcal{E}\{e/w\}, \mathcal{F}' = \mathcal{F}\{f/z\}$$

$$(\mathcal{E}, \#\mathsf{l}\ x; Q, H[a : \oplus_{\ell \in L} A_\ell\langle q, \mathcal{G}, P\rangle b : B]) \Mapsto (\mathcal{E}, Q, H[a : A_{\#\mathsf{l}}\langle q @ \#\mathsf{l}, \mathcal{G}, P\rangle b : B])^{wr} \qquad [\text{S}\oplus]$$
$$a = \mathcal{E}(x)$$

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

$$(\mathcal{E}, \mathsf{cut!}\ \{y.R \mid !x{:}A \mid P\}, H) \Mapsto (\mathcal{G}, P, H) \qquad [\text{SCut!}]$$
$$\mathcal{G} = \mathcal{E}\{\mathsf{clos!}(y{:}A, \mathcal{E}, R)/x\}$$

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

$$(\mathcal{E}, ?y; Q, H[a{:}\emptyset\langle \mathsf{clos!}(z{:}B, \mathcal{F}, R), \emptyset, 0\rangle b{:}?B]) \Mapsto (\mathcal{G}, Q, H) \qquad [\text{S?}]$$
$$b = \mathcal{E}(y), \mathcal{G} = \mathcal{E}\{\mathsf{clos!}(z{:}B, \mathcal{F}, R)/y\}$$

$$(\mathcal{E}, \mathsf{call}\ y(w); Q, H) \Mapsto (\mathcal{E}', Q, H[a{:}A\langle \mathsf{nil}, \mathcal{F}', R\rangle b{:}\overline{A}])^{p} \qquad [\text{Scall}]$$
$$a, b = \text{new}, \mathcal{E}' = \mathcal{E}\{a/w\}, \mathcal{F}' = \mathcal{F}\{b/z\}, \mathsf{clos!}(z : \overline{A}, \mathcal{F}, R) = \mathcal{E}(y)$$

$$N.B. : (a{:}A\langle q, \mathcal{G}, Q\rangle b{:}B)^{rw} \triangleq \text{if } q = \mathsf{nil} \text{ then } b{:}B\langle q, \mathcal{G}, Q\rangle a{:}A \text{ else } a{:}A\langle q, \mathcal{G}, Q\rangle b{:}B$$

$$(\mathcal{E}, P, H[a{:}A\langle q, \mathcal{G}, Q\rangle b{:}B])^{wr} \triangleq \text{if } A{+} \text{ then } (\mathcal{E}, P, H[a{:}A\langle q, \mathcal{G}, Q\rangle b{:}B]) \text{ else } (\mathcal{G}, Q, H[a{:}A\langle q, \mathcal{E}, P\rangle b{:}B])$$

$$(\mathcal{E}, P, H[a{:}A\langle \mathsf{nil}, \mathcal{G}, Q\rangle b{:}B])^{p} \triangleq \text{if } A{+} \text{ then } (\mathcal{E}, P, H[a{:}A\langle \mathsf{nil}, \mathcal{G}, Q\rangle b{:}B]) \text{ else } (\mathcal{G}, Q, H[b{:}B\langle \mathsf{nil}, \mathcal{E}, P\rangle a{:}A])$$

Fig. 18. The Linear SAM Transition Rules (Full Classical Linear Logic with Exponentials).

We now discuss the SAM rules for the exponentials ([S!], [S?], [Scall+] and [Scall-]). Values of exponential type are represented by exponential closures $\mathsf{clos!}(z, \mathcal{F}, R)$. Recall that a session type may terminate in either $\mathbf{1}$, $\bot$ or in a exponential $!A/?A$ (cf. 4.7). 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. 10). The execution rule [S?] is similar to rule [S$\parr$]: it pops a closure from the queue (which, in this case, becomes empty), and instead of using it immediately, adds it to the environment to become persistently available (cf. reduction rule [S?] Fig. 10).

$$(\mathcal{E}, \mathsf{sendty}\ x(T); Q, H[a{:}\exists X.A\langle q, \mathcal{G}, P\rangle b{:}B]) \Rrightarrow (\mathcal{E}, Q, H[a{:}\{T/X\}A\langle q@\mathsf{ty}(T), \mathcal{G}, P\rangle b{:}B])^{wr} \qquad [\mathrm{S}\exists]$$
$$a = \mathcal{E}(x)$$

$$(\mathcal{E}, \mathsf{recvty}\ y(X); Q, H[a{:}A\langle \mathsf{ty}(T)@q, \mathcal{G}, P\rangle b{:}\forall X.B]) \Rrightarrow (\mathcal{E}, \{T/X\}Q, H[(a{:}A\langle q, \mathcal{G}, P\rangle b{:}\{T/X\}B)^{rw}]) \qquad [\mathrm{S}\forall]$$
$$b = \mathcal{E}(y)$$

$$(\mathcal{E}, \mathsf{unfold}_{\mu}\ x; Q, H[a{:}\mu X.A\langle q, \mathcal{G}, P\rangle b{:}B]) \Rrightarrow (\mathcal{G}, P, H[a{:}\{\mu X.A/X\}A\langle q@\mathsf{step}, \mathcal{E}, Q\rangle b{:}B]) \qquad [\mathrm{S}\mu]$$
$$a = \mathcal{E}(x)$$

$$(\mathcal{E}, \mathsf{rec}\ Y(u, \vec{w}); Q\ [y, \vec{z}], H[a{:}A\langle \mathsf{step}, \mathcal{G}, P\rangle b{:}\nu X.B]) \Rrightarrow$$
$$(\mathcal{G}, P, H[a{:}A\langle \mathsf{nil}, \mathcal{F}, \{(\mathsf{rec}\ Y(u, \vec{w}); Q)/Y\}Q\rangle b{:}\{\nu X.B/X\}B])^{p} \qquad [\mathrm{S}\nu\nu]$$
$$b = \mathcal{E}(y), \mathcal{F} = \mathcal{E}\{\mathcal{E}(y)/u\}\{\mathcal{E}(\vec{z})/\vec{w}\}$$

Fig. 19. The Linear SAM Transition Rules (Polymorphism and Recursion)

Any environment stored closure holding a replicated process may be called by client code via the transition rules [Scall], which for each invocation create a new linear session object to be composed (passed to) the client code. Thus, rule [Scall] operates in a similar way to rule [S⅋], instead of activating a linear closure popped from the queue, it activates an replicable closure fetched from the environment. The need for adjusting the endpoint of the newly created linear session, depending on the polarity of the type of the new session endpoint name w. We implement this polarity dependence using the two variant rules [Scall+] and [Scall-] as already discussed in Section 3 for the case of [S⅋]. Notice that the process closure passed in send and receive ([S⊗] and [[S⅋]) are linear linearly, thus transfered with unique ownership from sender to queue and from queue to receiver without any need to be stored in the environment, unlike replicated closures, which are potentially shareable in client code. The [SCut!] rule simply binds the cut!-bound replicated closure bound in the environment, and proceeds with the execution of P.

The SAM rules for the polymorphic types and recursion are presented in Figure 19. For the case of type send and receive, types are passed around with type values $\mathsf{ty}(T)$, where T is a closed type. Notice the type substitution applied to the continuation Q, in [S∀].

Given the analogy of the recursion-related rules [Sμ] and [Sν] with the related CLLB rules [μ] and [ν], they barely deserve any special remark. Notice however how the SAM uses the environment to bind arguments to parameters in the rule for the co-recursive call [Sν]. We now proceed our technical development and update our meta-theoretical results for the environment-based Linear SAM.

6.1 Correctness of the Environment-based Linear SAM

The proofs of soundness and progress for the environment-based linear SAM follow the pattern of our development in Section 5.2. We first reformulate the notion of configuration readiness (c.f. Definition 5.2) to express the SAM invariants now required to deal with the presence of environments, closures, and exponential constructs.

We say an environment $\mathcal{G}$ covers a process P is $dom(\mathcal{G}) \subseteq \mathsf{fn}(P)$.

Definition 6.1 (Modes). In a well-moded session record $a{:}A\langle q, \mathcal{G}, R\rangle b{:}B$, $\mathcal{G}$ covers R and at most one of a, b may occur free in $\mathcal{G}(R)$. Moreover, a well-moded session record is in:

(1) write-mode, if $b \in \mathsf{fn}(\mathcal{G}(R))$, A is positive and $\mathsf{step} \notin q$.

(2) read-mode, if $a \in \mathsf{fn}(\mathcal{G}(R))$ or $R = \mathbf{0}$ and either

 (a) A is not positive and $\mathsf{step} \notin q$, or

(b) q is step-terminated.

Definition 6.2 (Ready Configuration). A SAM configuration $(\mathcal{E}, P, H)$ is ready if $\mathcal{E}$ covers P and H, and any session record in H is either in write-mode or read-mode.

Definition 6.3 handles encodings of CLLB processes into environment-based SAM configurations, along the lines of Definition 5.3 but now dealing with environments. We first introduce the following notations:

Write $\mathcal{E} \approx_P \mathcal{G}$ if $\mathcal{G}(x) = \mathcal{E}(x)$ for all of $x \in \mathsf{fn}(P)$.

Write $\mathcal{E} \approx_{z.P} \mathcal{G}$ if $\mathcal{G}(x) = \mathcal{E}(x)$ for all of $x \in \mathsf{fn}(P) \setminus z$.

Let q be a CLLB queue and q a (environment-based) SAM queue.

Write $qe \approx q^{\mathcal{G}}$ if $qe_i = \mathsf{clos}(z, \mathcal{F}, R)$ (resp. $\mathsf{clos!}(z, \mathcal{F}, R)$) iff $q_i = \mathsf{clos}(z, R)$ (resp. $\mathsf{clos!}(z, R)$) and $\mathcal{F} \approx_{z.R} \mathcal{G}$.

Definition 6.3 (Encoding CLLB *to full SAM).* For environments $\mathcal{E}, \mathcal{E}'$, well-typed CLLB processes P, P' and well-formed heaps H, H' let relation $(\mathcal{E}, P, H) \overset{\text{ence}}{\Mapsto} (\mathcal{E}, P', H')$ be defined by the rules.

$$(\mathcal{E}, \mathsf{cut}\ \{P(x)\ |\overline{x}{:}A\ [q]\ y{:}B|\ Q(y)\}, H) \overset{\text{ence}}{\Mapsto} (\mathcal{G}, P(x), H[a{:}A\langle qe, \mathcal{F}, Q(y)\rangle b{:}B])\ [\text{EC+}] \quad (a\,\langle-\rangle\, b \text{ write-mode})$$

$$(\mathcal{E}, \mathsf{cut}\ \{P(x)\ |\overline{x}{:}A\ [q]\ y{:}B|\ Q(y)\}, H) \overset{\text{ence}}{\Mapsto} (\mathcal{F}, Q(y), H[a{:}A\langle qe, \mathcal{G}, P(x)\rangle b{:}B])\ [\text{EC-}] \quad (a\,\langle-\rangle\, b \text{ read-mode})$$

where $\mathcal{F} \approx_{Q(y)} \mathcal{E}\{b/y\}$ and $\mathcal{G} \approx_{P(x)} \mathcal{E}\{a/x\}$ and $qe \approx q^{\mathcal{E}}$

$$(\mathcal{E}, \mathsf{cut!}\ \{y.R\ |!x : A|\ P\}, H) \overset{\text{ence}}{\Mapsto} (\mathcal{E}\{\mathsf{clos!}(y, \mathcal{F}, R)/x\}, P, H) \quad [\text{E!}]$$

where $\mathcal{F} \approx_{y.R(y)} \mathcal{E}$

Given $P \vdash^{\mathsf{B}} \emptyset; \emptyset$, we write $P \overset{\text{ence}*}{\Mapsto} C$ for $(\emptyset, P, \emptyset) \overset{\text{ence}*}{\Mapsto} C$ and $ence(P) = C$ for $P \overset{\text{ence}*}{\Mapsto} (\mathcal{E}, A, H)$, where $C = (\mathcal{E}, A, H)$ for $\Delta \vdash_{\mathsf{CLL}} A$ with $A = \mathcal{A}$ or $A = 0$.

The encoding introduces environments in session records and closures that (over)approximate all environments possibly generated in a computation (via the equivalences $\mathcal{E} \approx_P \mathcal{G}$). This strengthened property, required by our correctness proofs, enables the encoding to emulate all possible closures created in histories prior to the current CLLB state $ence(P)$, which may contain irrelevant bindings. We can show

Lemma 6.4 (Readiness of $\mathcal{E}$-Encoding). *Basic properties of* $\overset{\text{ence}}{\Mapsto}$.

1. *If C is ready and $C \overset{\text{ence}}{\Mapsto} \mathcal{D}$, then $C \Mapsto \mathcal{D}$ with $\mathcal{D}$ is ready.*
2. *If $(\mathcal{E}, P, H)$ is ready and $P \vdash_{\mathsf{CLL}} \Delta$ then $(\mathcal{E}, P, H) \overset{\text{ence}*}{\Mapsto} (\mathcal{E}', A, H') = C$ and $(\mathcal{E}, P, H) \overset{\text{cut}*}{\Mapsto} C$, for $A = \mathcal{A}$ or $A = 0$.*
3. *If $P \vdash_{\mathsf{CLL}} \emptyset$ then $ence(P)$ is ready.*

Proof. (1) Any $\overset{\text{enc}}{\Mapsto}$ step in a ready configuration creates a new session record in either write- or read-mode. (2) For a CLL process P, $(\mathcal{E}, P, H) \overset{\text{ence}}{\Mapsto} C$ must be by [Cut-write], matching [SCut]. (3) By (1,2) applied to $(\emptyset, P, \emptyset) \overset{\text{enc}*}{\Mapsto} enc(P)$. □

A SAM configuration $C = (\mathcal{E}, P, H)$ is live if $P = \mathcal{A}$.

Lemma 6.5 (SAM-CLLB $\mathcal{E}$-Step Safety). *Let $P \vdash^{\mathsf{B}} \emptyset$ and $ence(P)$ a ready SAM configuration. If $ence(P)$ is live then (1) there is C ready such that $ence(P) \Mapsto C$ and (2) there is Q such that $P \to^{\mathsf{B}} Q$ and $Q \overset{\text{ence}*}{\Mapsto} C \overset{\text{cut}*}{\Mapsto} enc(Q)$.*

Proof. See Appendix 10.4, we detail the cases for exponentials. Similar to the proof of Lemma 5.5. □

Theorem 6.6 (SAM Soundness wrt CLLB). *Let $P \vdash^{\mathsf{B}} \emptyset; \emptyset$ and $ence(P)$ ready.*
If $ence(P) \overset{*}{\Mapsto} C$ then there is Q such that $P \Rightarrow^{\mathsf{B}} Q$ and $Q \overset{\text{ence}*}{\Mapsto} C$.

Proof. Similar to the proof of Lemma 5.6, using Lemma 6.5. □

Theorem 6.7 (SAM Soundness wrt CLL). *Let* $P \vdash \emptyset; \emptyset$.
If $(\emptyset, P, \emptyset) \overset{*}{\Mapsto} C$ *then there is* Q *such that* $P \Rightarrow_{\mathsf{CLL}} Q$ *and* $Q^{\dagger} \Rightarrow^{\mathsf{Bap}} \overset{\mathsf{ence}*}{\Mapsto} C$.

Proof. Similar to the proof of Lemma 5.7, but using Theorem 6.6. □

An environment-based SAM configuration C live if $C = (\mathcal{E}, P, H)$ if $P \neq 0$.

Theorem 6.8 (SAM Progress). *Let* $P \vdash_{\mathsf{CLL}} \emptyset$. *If* $(\emptyset, P, \emptyset) \overset{*}{\Mapsto} C$ *then either* $C = (0, \emptyset)$ *or there is* C' *such that* $C \Mapsto C'$.

Proof. Similar to Theorem 5.8, but using Lemma 6.5. □

7 CONCURRENT SEMANTICS

We have argued that our design for the SAM provides an effective deterministic execution model for the implicitly sequential session-typed program idioms. Of course, in general one often want parts of programs to be executed concurrently, at various levels of granularity. In this section, we demonstrate how the Linear SAM supports the ability to naturally segregate and coordinate, at a fine-grain, both sequential and concurrent behaviours, as generally supported by the session calculus. We claim that the SAM provides a principled basis to approach a unified execution model on which essentially sequential parts of session-based programs may be efficiently executed by the deterministic application of SAM transitions, without concurrent synchronisation mechanisms, while explicitly parallel and concurrent execution and may be selectively used whenever necessary, for performance or functional requirements.

In this Section we discuss the extension of the Linear SAM from a single-threaded machine to a multi-threaded machine that supports the concurrent execution of CLL processes composed by concurrent versions of mix and cut, semantically equivalent to the basic cut and mix constructs in all accounts (typing rules and conversions / reductions), but where the composed processes execute in separate threads.

$$P \quad ::= \quad \ldots \quad | \quad \mathsf{ccut}\ \{P \mid x : A \mid Q\} \quad | \quad \mathsf{cpar}\ \{P \mid\mid Q\}$$

To facilitate our presentation, we revert to the basic SAM introduced in Section 3 without environments and exponentials, since such features are orthogonal to our focus on concurrency. This allows us to develop the concurrent SAM as an extra modular layer on top of the basic architecture and transition semantics, which we keep untouched.

The fundamental step, presented in Figure 20, consists in extending configurations from pairs process/heap (P, H) to pairs process-multiset/heap $(\mathcal{M}, H)$, where $\mathcal{M}$ is a multiset of processes $P_{i \in I}$. Intuitively, each process P_i in $\mathcal{M}$ is executing in a separate, concurrent thread, thus $\mathcal{M}$ plays the role of a thread pool.

Moreover, the concurrent SAM architecture adds concurrent session records, noted $a\,\langle q \rangle\,b$, to the basic session records of the SAM as defined in Section 5. While basic session records support sequential session interaction using co-routining, concurrent session records support concurrent (and/or parallel) session interaction. A concurrent session record corresponds essentially to a bidirectional concurrent message queue q, asynchronously accessed by session-interacting processes via the endpoints a and b. Each individual thread $P_i \in \mathcal{M}$ executes locally according to the Linear SAM sequential and deterministic semantics presented in prior Sections, executing actions on sequential session record endpoints, until a concurrent process action, that is, an action (send / receive / selection / etc.) on an endpoint of a concurrent queue, is reached.

$$
\begin{array}{lcll}
\mathcal{M} & = & \{P_1, P_2, \ldots, P_n\} & \text{Process Multiset)} \\
S & ::= & (\mathcal{M}, H) & \text{(Concurrent SAM Configuration)} \\
\mathit{SessionRec} & ::= & x\langle q, P\rangle y & \text{(Sequential Session Record)} \\
 & | & x\,\langle q\rangle\, y & \text{(Concurrent Session Record)}
\end{array}
$$

Fig. 20. The Concurrent SAM.

$$
\frac{(P, H) \Mapsto (P', H')}{(P \uplus \mathcal{M}, H) \Mapsto^{c} (P' \uplus \mathcal{M}, H')}\ \text{[Srunc]} \qquad \frac{(\mathcal{M}, H) \Mapsto^{c} (\mathcal{M}', H')}{(0 \uplus \mathcal{M}, H) \Mapsto^{c} (\mathcal{M}', H')}\quad \text{[S0p]}
$$

$$
(\mathsf{ccut}\ \{P \mid x : A^{+} \mid Q\} \uplus \mathcal{M}, H) \Mapsto^{c} (\{P, Q\{y/x\}\} \uplus \mathcal{M}\}, H[x : A\,\langle \mathsf{nil}\rangle\, y : \overline{A}]) \quad \text{[SCutp]}
$$

$$
((P \mid\mid Q) \uplus \mathcal{M}, H) \Mapsto^{c} (\{P, Q\} \uplus \mathcal{M}\}, H) \quad \text{[SMixp]}
$$

Fig. 21. Transition rules for Concurrent SAM configurations

We define in Figure 21 the transition system for the execution relation of concurrent SAM configurations, represented by $C \Mapsto^{c} C'$. It is essentially defined as an extra layer on top of the rules for the sequential system, that expresses execution for each single thread. Concurrency is modelled by the non-deterministic interleaving of atomic steps from each sequential thread. The basic SAM execution rule for a single thread, based on $\mathcal{S} \Mapsto \mathcal{S}'$, is now used to define a one step execution on a concurrent SAM configuration, as expressed by rule [Srunc]; it non-deterministically picks a (ready) process P in the multiset and performs one transition step on P using the basic transition system of Section 5.

Notice that a single common heap is shared by all processes in $\mathcal{M}$. As shown below, the CLL type system ensures that all session record usages are safely used by the concurrent running processes, due to the linear typing. Rule [S0p] clears a terminated thread, while rules [SMixp] and [SCutp] forks the current thread into two new threads. For [SCutp] a new concurrent session record is created, connecting the two processes in the queue, while for [SMixp] the threads are set to simply run in parallel. Notice that, remarkably, our uniform logical foundation naturally supports concurrent versions of cut and mix to freely and safely combine with their sequential versions, since they all satisfy the same congruence rules, even if executed under different strategies. As an example, consider the processes

$$
\mathsf{cut}\ \{P \mid x : A \mid \mathsf{cut}\ \{Q \mid y : A \mid R\}\} \quad \mathsf{ccut}\ \{P \mid x : A \mid \mathsf{cut}\ \{Q \mid y : A \mid R\}\} \quad \mathsf{cut}\ \{P \mid x : A \mid \mathsf{ccut}\ \{Q \mid y : A \mid R\}\}
$$

which are convertible modulo structural equivalence, but implement different execution strategies. On the left, the whole process is executed using the SAM sequential strategy. In the middle P and $\mathsf{cut}\ \{Q \mid y : A \mid R\}$ are executed concurrently via busy waiting on messages exchanged in x, while Q and R execute sequentially by co-routining, while on the right the converse happens.

Concurrent process actions on concurrent queues are assumed to be implemented as atomic isolated transactions. In the operational semantics, concurrent actions are defined in a way similar to the "in thread" sequential ones, except that while in the sequential semantics computation we know that negative (read) operations are only performed on non-empty queues (session records in read-mode), in the concurrent semantics that is not the case, hence negative actions (e.g., session receive) are blocking. More concretely, while concurrent positive actions (e.g., session send) always progress by writing a value into the write endpoint of the queue, concurrent negative actions will either read off a value from the queue if such value is available, or block, waiting for a value to become available. However, we may

$$
\begin{array}{ll}
(\mathsf{close}\ x, H[x\,\langle q\rangle\,y]) \Mapsto (0, H[x\,\langle q@\checkmark\rangle\,y]) & [\mathrm{S}\mathbf{1}\mathrm{c}] \\
(\mathsf{wait}\ y; P, H[x\langle\checkmark, y\rangle]) \Mapsto (P, H) & [\mathrm{S}\bot\mathrm{c}] \\
(\mathsf{send}\ x(z.R); Q, H[x\,\langle q\rangle\,y]) \Mapsto (Q, H[x\,\langle q@\mathsf{clos}(z,R)\rangle\,y]) & [\mathrm{S}\otimes\mathrm{c}] \\
(\mathsf{recv}\ y(w); Q, H[x\,\langle \mathsf{clos}(z,R)@q\rangle\,y]) \Mapsto (R, H[x\,\langle q\rangle\,y^{rw}][z\langle \mathsf{nil}, Q\rangle w])^{p} & [\mathrm{S}⅋\mathrm{c}] \\
(\mathsf{fwd}\ x\ y, H[z\,\langle q_1\rangle\,x][y\,\langle q_2\rangle\,w]) \Mapsto (0, H[z\,\langle q_2@q_1\rangle\,w]) & [\mathrm{SfwdLc}] \\
(\mathsf{fwd}\ x\ y, H[z\,\langle q_1\rangle\,x][y\langle q_2, Q\rangle w]) \Mapsto (Q, H[z\,\langle q_2@q_1\rangle\,w]) & [\mathrm{SfwdMc}] \\
(\mathsf{fwd}\ x\ y, H[z\langle q_1, Q\rangle x][y\,\langle q_2\rangle\,w]) \Mapsto (Q, H[z\,\langle q_2@q_1\rangle\,w]) & [\mathrm{SfwdMc}]
\end{array}
$$

$$
\begin{array}{l}
a\,\langle q\rangle\,b^{rw} \triangleq \text{if } (q = \mathsf{nil}) \text{ then } b\,\langle q\rangle\,a \text{ else } a\,\langle q\rangle\,b \\
(P, H[x{:}A\,\langle \mathsf{nil}\rangle\,y{:}B])^{p} \triangleq \text{if } (A+) \text{ then } (P, H[x{:}A\,\langle \mathsf{nil}\rangle\,y{:}B]) \text{ else } (P, H[y{:}B\,\langle \mathsf{nil}\rangle\,x{:}A])
\end{array}
$$

Fig. 22. Additional SAM Transition Rules for concurrent actions (sample)

prove (Theorem 7.6) that in our typed language any such waiting processes will always progress , since the process holding the dual endpoint will eventually write the expected value in the queue. The technical development and results presented below make this claim precise. We present in Figure 22 the concurrent SAM transition rules for $\mathbf{1}$,$\bot$,$\otimes$,⅋ typed actions, it should be clear how to define rules for other actions, since they follow the same pattern. When the endpoints in as forwarder refer to session records of the same kind (concurrent or sequential), forwarding is implemented by queue merging, yielding a single session record of the given kind. In the case for forwarding between session records of different kinds (one sequential and the other concurrent), the merge results into a concurrent session record. These rules extend the basic transition system of Figure 14, invoked by the premise of rule [SRunc], and apply to single thread execution. To define readiness for concurrent configurations we essentially need to require readiness for every thread.

Definition 7.1 (Ready). Configuration $\mathcal{S} = (\mathcal{M}, H)$ is ready if for all $P \in \mathcal{M}$ the configuration $C_P = (P, H)$ is ready.

Notice that readiness for concurrent session records does not involve read/write modes, required in the sequential case to ensure progress. We now define an appropriate encoding $\overset{\mathsf{encc}}{\Mapsto}$ of concurrent CLLB processes into concurrent SAM states. The definition relies on $enc(P, H)$ (Definition 5.3) in the base case [Thr] to encode a single thread.

Definition 7.2 (Encoding CLLB *to CSAM).* For multisets $\mathcal{M}, \mathcal{M}'$ of well-typed CLLB processes and well-formed heaps H, H' let relation $(\mathcal{M}, H) \overset{\mathsf{encc}*}{\Mapsto} (\mathcal{M}', H')$ be defined by the rules.

$$
\begin{array}{lrl}
(\{0\} \uplus \mathcal{M}, H) \overset{\mathsf{encc}}{\Mapsto} (\mathcal{M}, H) & & [\mathrm{C}0] \\
(\{P\} \uplus \mathcal{M}, H') \overset{\mathsf{encc}}{\Mapsto} (\{Q\} \uplus \mathcal{M}, H') & \text{if } enc(P,H) \overset{\mathsf{enc}}{\Mapsto} (Q, H') & [\mathrm{CThr}] \\
(\mathsf{cpar}\ \{P \mid\mid Q\}\ \uplus \mathcal{M}, H) \overset{\mathsf{encc}}{\Mapsto} (\{P, Q\} \uplus \mathcal{M}, H) & & [\mathrm{CMix}] \\
(\mathsf{ccut}\ \{P \mid \overline{x} : A[q]y : B \mid Q\} \uplus \mathcal{M}, H) \overset{\mathsf{encc}}{\Mapsto} (\{P, Q\} \uplus \mathcal{M}, H[x\,\langle q\rangle\,y]) & & [\mathrm{CCut}]
\end{array}
$$

Given $P \vdash^{\mathsf{B}} \emptyset$, write $P \overset{\mathsf{encc}*}{\Mapsto} C$ for $(P, \emptyset) \overset{\mathsf{encc}*}{\Mapsto} C$ and $encc(P) = C$ for $P \overset{\mathsf{encc}*}{\Mapsto} (\tilde{A}, H)$, $\Delta_i \vdash_{\mathsf{CLL}} A_i$ with $A_i = \mathcal{A}$.

If $encc(P) = (\tilde{A}, H)$ the active multiset $\tilde{A}$ only contains action processes $\mathcal{A}$ ready for execution or is empty. We expected the following basic sanity properties (cf. Lemma 5.4). We write $C \overset{\mathsf{cut}*}{\Mapsto} \mathcal{D}$ if $C \overset{*}{\Mapsto} \mathcal{D}$ by repeated use of Concurrent SAM cut transitions [CCutp], [CMixp], [C0p], and [SCut] (via [Srunc]).

Lemma 7.3 (Readiness of Encoding). *Basic properties of* $\overset{\mathsf{encc}}{\Mapsto}$.

(1) *If* C *is ready and* $C \overset{\mathsf{encc}}{\Mapsto} \mathcal{D}$, *then* $C \Mapsto \mathcal{D}$ *with* $\mathcal{D}$ *is ready.*

(2) *If* $(\mathcal{M}, H)$ *is ready and* $\mathcal{M} \vdash_{\mathsf{CLL}} \Delta$ *then* $(\mathcal{M}, H) \overset{\mathsf{encc}*}{\Mapsto} (\tilde{A}, H') = C$ *and* $(\mathcal{M}, H) \overset{\mathsf{cut}*}{\Mapsto} C$.

(3) *If* $P \vdash_{\mathsf{CLL}} \emptyset$ *then* $encc(P)$ *is ready.*

Proof. Similar to the proof of Lemma 5.4. □

A concurrent SAM configuration C is live if $C = (\mathcal{M}, H)$ with $\mathcal{A} \in \mathcal{M}$. We have

Lemma 7.4 (CSAM-CLLB Step Safety). *Let* $P \vdash^{\mathsf{B}} \emptyset$ *and* $encc(P)$ *a ready SAM configuration. If* $encc(P)$ *is live then (1) there is* C *ready such that* $ence(P) \Mapsto C$ *and (2) there is* Q *such that* $P \rightarrow^{\mathsf{B}} Q$ *and* $Q \overset{\mathsf{encc}*}{\Mapsto} C \overset{\mathsf{cut}*}{\Mapsto} ence(Q)$.

Proof. See Appendix 10.5. The proof modularly combines progress for the sequential SAM (cf. Lemma 5.5) with the proof for general progress for CLLB, which is based on the technique of inductive observations (cf. Lemma 4.9). □

We build on the strong properties identified in main Lemma 7.4 to state the following soundness results for the concurrent SAM. The proofs closely follow the structure developed in Section 5 for the basic Linear SAM.

Theorem 7.5 (CSAM Soundness wrt CLL). *Let* $P \vdash_{\mathsf{CLL}} \emptyset$.
If $(\{P\}, \emptyset) \overset{*}{\Mapsto} C$ *then there is* Q *such that* $P \Rightarrow_{\mathsf{CLL}} Q$ *and* $Q^{\dagger} \Rightarrow^{\mathsf{Bap}} \overset{\mathsf{encc}*}{\Mapsto} C$.

Proof. Based on Lemma 7.4, follow the proof steps of Theorem 5.7. □

Theorem 7.6 (CSAM Progress). *Let* $P \vdash_{\mathsf{CLL}} \emptyset$. *If* $(\{P\}, \emptyset) \overset{*}{\Mapsto} C$ *then either* $C = (\emptyset, \emptyset)$ *or there is* C' *such that* $C \Mapsto C'$.

Proof. Based on Lemma 7.4, follow the proof steps of Theorem 5.8. □

The concurrent 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. Remarkably, the whole model is expressed in the common language of linear logic, statically ensuring safety, proper resource usage, termination, and deadlock absence by static typing. In particular, Theorem 7.6 states that any well-typed will either progress or terminate in a leak-free configuration, where all heap records have been deallocated and all concurrent threads have terminated. In the next section, we discuss an artifact implementation of the SAM, some experimental results obtained, and discuss some possible pragmatic extensions.

8 IMPLEMENTATION AND EXPERIMENTAL RESULTS

In this section, we briefly describe our current prototype implementation of the SAM[1]. It is available in open source, and provided as an alternative backend execution engine for our ongoing CLASS language development [18, 66–69], which provides a static type-checker and execution environment for CLASS programs. CLASS is a linear session-based language whose core includes CLL as presented in this paper, and that flexibly supports realistic concurrent programming idioms. Its linear logic based type system statically ensures that programs never misuse or leak stateful resources or memory, they never deadlock, and they always terminate. The current version efficiently provides basic datatypes (integers, booleans, strings), ML-style let expressions and conditionals, polymorphism, and recursive and co-recursive types. The system also support a form of shareable mutable state [65, 66, 68], not considered in this paper.

[1] The most recent version of the implementation with the examples of this paper may be found in [19].

These first interpreters [18, 69] adopt a pervasive concurrent model of sessions, based on the java.util.concurrent.* package, using fine-grained threading and locking to support synchronous session interaction. The work in this paper was strongly motivated by experimentation with these implementations, and questioning whether an important bulk of the computations could be efficiently sequentialised.

The implementation of the SAM discussed here covers the full basic CLL language as presented in Section 6 of this paper, but leaves for future work the development of appropriate support for other features of CLASS. Nevertheless, our SAM implementation already supports the basic CLASS datatypes and associated operations, and demonstrates the practical feasibility of the model developed in the present work.

The machine is implemented as an interpreter that closely follows the architecture and transition system for the environment-based SAM (Section 6). It deterministically manipulates machine configurations in a main loop that at each iteration executes the transition associated to each program construct in the current environment and heap.

A machine state configuration is thus represented by the structure SAMState (Figure 23 (top)), that aggregates the current code and environment(s), and is imperatively updated by the machine transitions. The SAM heap is not explicitly represented in the SAMState, as session records are implicitly represented as host language objects. Queues on session records are efficiently implemented with arrays and updated in place, where queue positions are directly indexed by integer offsets determined at type-checking from the (session) types. This also allows us to statically determine the size of session records from types, including in the presence of recursion, where the record space gets conveniently recycled at unfold steps.

The SessionField slots[] field stores up to int size queue values during session execution. The slots[] indices are used linearly, written once during a sequence of operations of positive polarity, and read once during the matching sequence of operations of negative polarity, where the read operation clears (sets to null) the session slot contents. The SessionClosure cont field stores a reference to the session "continuation" (suspended) process code P, it may be understood by analogy to the "return address" of the activation record in a functional language implementation. The SessionClosure cont field also holds the session record environment $\mathcal{E}$, which would then correspond to the "dynamic link" of the activation record in a functional language implementation as well. Session record endpoints are represented by the structure IndexedSessionRef, that bundles a session record reference with the current write/read int offset in the slots[] array, for technical reasons we prefer to store the currently active polarity in the boolean polarity field of the session record. Notice that the the session offset is reset to 0 whenever a session moves from read mode to write mode (empty queue), this allows the size of the slots[] array to be statically fixed as the length of the longer positive section of its session type.

As a consequence of the linear typing discipline, the SAM promises a very efficient memory management scheme, which we already approximate in our prototype. In particular, linear session records and associated environment entries are explicitly allocated when sessions are created, and deallocated (in fact, recycled) after the session close/waits, without any need for garbage-collection.

The toplevel REPL of the interpreter provides a trace option that lists the SAM execution. For instance, for the example in Section 3.2, rewritten in CLASS source, we obtain the execution trace depicted in Figure 23 (middle). Each line shows the SAM operation executed, the id of the active session record, and the current offset.

It is interesting to consider the execution metrics for some toy examples, comparing the fully concurrent execution of [18, 69] with the more efficient linear SAM execution, as depicted in Figure 23 (bottom). We selected five benchmark code samples. The SpeedUp column indicates the execution time ratio between the SAM execution and the basic fully concurrent implementation. Not surprisingly even our fairly inefficient SAM interpreter may improve runtimes in 1-2

```
SAMState {
    ASTNode code;
    Env<SessionField> env;

}
```

```
SessionRecord {
    int size;
    SessionField slots[];
    SessionClosure cont;
    boolean polarity;
}
```

```
IndexedSessionRef
extends SessionField {
    SessionRecord srec;
    int offset;
}
```

```
SessionClosure
extends SessionField {
    String objectid;
    ASTNode code;
    Env<SessionField> env;
}
```

(selected data structures)

```
type T1 {
    recv ~ lint; recv ~ lint; close
};;

type T2 {
    send lint; recv ~ lint; close
};;

proc P1(b: T1 ) {
  recv b(x); recv b(z); close b
};;

proc Q1(d:T2, b: ~ T1) {
  send d(1); send b(3); fwd d b
};;

proc R1(d: ~ T2) {
  d(y); send d(2); wait d;()
};;

proc example2() {
  cut {
    P1(b) | b:~ T1 | Q1(d,b) | d:~ T2 | R1(d)
  }
};;
```

```
> trace 1;;
> sam example2();;
id-op example2
cut-op b SessionRecord@3d494fbf size=3
cut-op d SessionRecord@79fc0f2f size=3
id-op Q1
send-op d SessionRecord@79fc0f2f @ 0
id-op R1
recv-op d SessionRecord@79fc0f2f @ 0
send-op d SessionRecord@79fc0f2f @ 0
send-op b SessionRecord@3d494fbf @ 0
fwd-op b d SessionRecord@3d494fbf SessionRecord@79fc0f2f
id-op P1
recv-op b SessionRecord@3d494fbf @ 0
recv-op b SessionRecord@3d494fbf @ 1
clos-op b SessionRecord@3d494fbf @ 0
wait-op d SessionRecord@3d494fbf @ 0
empty-op
```

(execution trace of sample code)

Benchmark	SpeedUp	MaxMem	MaxMReqs
`systemF`	×54,4	771(4)	1527k
`bitcounter`	×19,7	12(5)	12
`ackermann`	×200,2	517(3)	2169k
`primesieve`	×158,5	502(5)	502
`cprimesieve`	×155,2	502(5)	502

Fig. 23. On the SAM Implementation.

orders of magnitude, by fully exploring the complete sequentiality present in the selected examples. The two other columns illustrate linear memory usage. The MaxMem column indicates the largest number of session records R of maximum length k for the specific benchmark (in terms of number of slots), listing the values as $R(k)$. For instance, a benchmark whose largest session records use 4 slots and which allocates 10 such (distinct) records would indicate $10(4)$ in its MaxMem column. The MaxMReqs column indicates the total number of times the records referred to by the MaxMem column were allocated.

The implemented benchmarks are as follows: `systemF`: This example explores a CLL encoding of recursive types using second-order types and parametricity, along the lines of Wadler's "recursion for free" [78, 81]. We encode naturals (church numerals) as polymorphic processes, as well as the maps needed to primitively represent the ADT functors, the associated fold and unfold operations, namely, we don't use the primitive SAM implementation of recursion, just linear logic with exponentials and second order quantifiers. Thus, this benchmark heavily exercises the basic session calculus. The benchmark repeatedly computes 4^4 and averages statistics.

`bitcounter`: We consider a CLL variant implementation of the process-based infinite precision bit counter from [75]. The benchmark repeatedly counts up to 2^{12} and averages statistics. Interesting to see the allocation of precisely $12 + 1$ session records of size 5, each representing one bit in the network.

`ackermann`: We recursively define the Ackermann function $Ack()$ over the naturals represented with recursive types (as in the example of Section 2.8). The benchmark repeatedly computes $Ack(6, 3) = 509$ and averages runtime statistics. Notice that, remarkably due to the implicit linear lazy computation strategy of the SAM, the required number of (three slot) session records representing zero and sucessor natural number nodes (517) essentially corresponds to the function output result.

`primesieve`: We consider a filter process network implementation of the sieve of Eratosthenes using the native integers and operations available in our SAM prototype. The benchmark repeatedly computes the first 500 primes, and averages runtime statistics. Again, interesting to see the allocation of no more than $500 + 2$ session records, each one essentially representing one filter for each prime stored in the pipeline.

`cprimesieve`: An example where the `primesieve` benchmark is spawned concurrently with a consumer that prints the received primes. This example illustrates the performance penalty of concurrent synchronizations when compared to the purely sequential formulation of the sieve.

A more detailed analysis of the many interesting issues related to an implementation of the SAM, in particular its potential for efficient compilation, is of course outside the scope of this paper, however, the discussion in this section already offers some perspective on the feasibility of the design to support the execution of realistic general higher-order linear session-based code.

9 CONCLUDING REMARKS AND RELATED WORK

We have introduced the Linear 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, including affine types and shared state [60, 68]. We have also produced a proof-of-concept implementation of the SAM, provided as an alternative backend execution engine for our ongoing CLASS language development [18, 66–69].

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 [48], 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 [24], Lafont's Linear Abstract Machine for the linear λ-calculus [46], and Krivine's Machine for the call-by-name λ-calculus [45] ; these works explored Curry-Howard correspondences to propose provably correct solutions. In [25], 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 [16, 83], 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 [37, 49, 60] played an important role to achieve a deterministic sequential reduction strategy for session-based programming, perhaps our main initial motivation. This allows the SAM to naturally and efficiently integrate the execution of sequential and concurrent session behaviours, as presented in Section 7, and suggests effective compilation schemes for mainstream virtual machines or compiler frameworks. Interestingly, in the expected encoding for the SAM of a function object of type $(\otimes A_i) \multimap B$ as a process of type $(\parr \overline{A_i}) \parr B$, the session record $x\langle q, \mathcal{E}, P\rangle y$ that mediates caller and callee essentially suggests a conventional stack frame where the queue q first stores the values passed and, after polarity inversion, the single returned value. Here, the continuation P represents the "function " return address, and the environment $\mathcal{E}$ the "static link". The SAM then also appears to conservatively extend familiar implementation structures for functional languages within a broader context.

The adoption of session and linear types is clearly increasing in research (e.g., [3, 27, 29, 60, 63, 68, 75, 85]) and general purpose languages (e.g., Haskell [9, 43], Rust [23, 47] Ocaml [40, 57], F# [56], Move [10], 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 realisation 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 [21], 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 [55] which develops a polarised variant of the $\bar{\lambda}\mu\tilde{\mu}$-calculus 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 [55] presents $\lambda\mu$-calculi featuring values and substitution of terms for variables (potentially deep within the term structure). Our system, being based on process calculus, features neither – there is no term representing the outcome of a computation, since computation is the interactive behaviour of processes (cf. game semantics); nor does computation rely on substitution in the same sense. Another significant distinction is that our work materialises 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.

We also note that it is not uncommon for works based on the correspondence between linear logic and session types [5, 6, 35] to adopt a semantics based on multiset rewriting [22] instead of the more traditional structural operational semantics (SOS) style used in process algebra. However, the multiset rewriting approach directly mirrors the SOS style and is immediately relatable to SOS [73]. This is in contrast with the semantics of the SAM whose correspondence with CLL is non-trivial, as illustrated by our technical development in Section 4.

In future work, we plan to study the semantics of the SAM in terms of games along the lines of [21, 24, 46]. We also intend to investigate the ways in which the evaluation strategy of the SAM can be leveraged to develop efficient compilation of fine-grained linear and session-based programming languages, and its relationship with effect handlers, coroutines and delimited continuations. On the way we will also extend the SAM to support shareable mutable state [65, 66, 68]. Linearity plays a key role in programming languages and environments for smart contracts in distributed ledgers [27, 71] manipulating linear resources (assets); it would be interesting to investigate how linear machines like the SAM would provide a basis for certified resource sensitive-computing [10, 86].

ACKNOWLEDGMENTS

We would like to thank the anonymous referees of [17], a preliminary and partial presentation of the Linear SAM. Work supported by NOVA LINCS (UIDB/ 04516/ 2020), INESC ID (UIDB/ 50021/ 2020), and BIG (Horizon EU 952226 BIG).

REFERENCES

[1] S. Abramsky. 1993. Computational Interpretations of Linear Logic. *Theoret. Comput. Sci.* 111, 1–2 (1993), 3–57.

[2] Samson Abramsky, Simon J Gay, and Rajagopal Nagarajan. 1996. Interaction categories and the foundations of typed concurrent programming. In *NATO ASI DPD*. 35–113.

[3] Bernardo Almeida, Andreia Mordido, Peter Thiemann, and Vasco T. Vasconcelos. 2022. Polymorphic lambda calculus with context-free session types. *Inf. Comput.* 289, Part (2022), 104948.

[4] Jean-Marc Andreoli. 1992. Logic Programming with Focusing Proofs in Linear Logic. *J. Log. Comput.* 2, 3 (1992), 297–347.

[5] Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 37 (2017), 29 pages.

[6] Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *Programming Languages and Systems (LNCS)*, Luís Caires (Ed.). Springer, 611–639.

[7] G. Bellin and P. Scott. 1994. On the π-Calculus and Linear Logic. *Theoret. Comput. Sci.* 135, 1 (1994), 11–65.

[8] P Nick Benton. 1994. A mixed linear and non-linear logic: Proofs, terms and models. In *International Workshop on Computer Science Logic*. Springer, 121–135.

[9] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.* 2, POPL (2018), 5:1–5:29.

[10] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, D. Russi, D. Sezer, T. Zakian, and R. Zhou. 2019. Move: A Language with Programmable Resources. (2019).

[11] Luís Caires and Jorge A. Pérez. 2017. Linearity, Control Effects, and Behavioral Types. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017 (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 229–259.

[12] Luís Caires and Jorge A. Pérez. 2017. Linearity, Control Effects, and Behavioral Types. In *Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201*. Springer-Verlag, Berlin, Heidelberg, 229–259.

[13] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2013. Behavioral Polymorphism and Parametricity in Session-Based Communication. In *Proceedings of the 22nd European Conference on Programming Languages and Systems* (Rome, Italy) *(ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 330–349.

[14] Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR 2010 - Concurrency Theory*, Paul Gastin and François Laroussinie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 222–236.

[15] Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Towards Concurrent Type Theory. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(TLDI '12)*. Association for Computing Machinery, New York, NY, USA, 1–12.

[16] Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear logic propositions as session types. *Mathematical Structures in Computer Science* 26, 3 (2016), 367–423.

[17] Luís Caires and Bernardo Toninho. 2024. The Session Abstract Machine. In *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024 (Lecture Notes in Computer Science, Vol. 14576)*, Stephanie Weirich (Ed.). Springer, 206–235.

[18] Luís Caires and Bernardo Toninho. 2024. The Session Abstract Machine (Artifact). (2024). https://doi.org/10.5281/zenodo.10459455

[19] Luís Caires and Bernardo Toninho. 2024. The Session Abstract Machine (Implementation). https://luiscaires.org/software/.

[20] Luca Cardelli. 1991. Typeful Programming. *IFIP State-of-the-Art Reports: Formal Description of Programming Concepts* (1991), 431–507.

[21] Simon Castellan and Nobuko Yoshida. 2019. Two sides of the same coin: session types and game semantics: a synchronous side and an asynchronous side. *Proc. ACM Program. Lang.* 3, POPL (2019), 27:1–27:29.

[22] Iliano Cervesato and Edmund S. L. Lam. 2015. Modular Multiset Rewriting. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9450)*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.). Springer, 515–531. https://doi.org/10.1007/978-3-662-48899-7_36

[23] R. Chen, S. Balzer, and B. Toninho. 2022. Ferrite: A Judgmental Embedding of Session Types in Rust. In *36th European Conference on Object-Oriented Programming, ECOOP 2022 (LIPIcs, Vol. 222)*, K. Ali and J. Vitek (Eds.). 22:1–22:28.

[24] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. 1987. The Categorical Abstract Machine. *Sci. Comput. Program.* 8, 2 (1987), 173–202.

[25] Olivier Danvy. 2004. A Rational Deconstruction of Landin's SECD Machine. In *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004 (LNCS, Vol. 3474)*, Clemens Grelck, Frank Huch, Greg Michaelson, and Philip W. Trinder (Eds.). Springer, 52–71.

[26] Ornela Dardha and Simon J. Gay. 2018. A New Linear Logic for Deadlock-Free Session-Typed Processes. In *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018 (LNCS, Vol. 10803)*, Christel Baier and Ugo Dal Lago (Eds.). Springer, 91–109.

[27] A. Das and F. Pfenning. 2022. Rast: A Language for Resource-Aware Session Types. *Log. Methods Comput. Sci.* 18, 1 (2022).

[28] Henry DeYoung, Luis Caires, Frank Pfenning, and Bernardo Toninho. 2012. Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication. In *Computer Science Logic*.

[29] Juliana Franco and Vasco Thudichum Vasconcelos. 2013. A Concurrent Programming Language with Refined Session Types. In *Software Engineering and Formal Methods - SEFM 2013 (LNCS, Vol. 8368)*, Steve Counsell and Manuel Núñez (Eds.). Springer, 15–28.

[30] Dan Frumin, Emanuele D'Osualdo, Bas van den Heuvel, and Jorge A. Pérez. 2022. A bunch of sessions: a propositions-as-sessions interpretation of bunched implications in channel-based concurrency. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 841–869.

[31] S. Gay and M. Hole. 2005. Subtyping for Session Types in the Pi Calculus. *Acta Informatica* 42, 2-3 (2005), 191–225.

[32] Simon Gay and Vasco Vasconcelos. 2010. Linear Type Theory for Asynchronous Session Types. *Journal of Functional Programming* 20, 1 (2010), 19–50.

[33] Jean-Yves Girard. 1991. A New Constructive Logic: Classical Logic. *Math. Struct. Comput. Sci.* 1, 3 (1991), 255–296.

[34] J.-Y. Girard. 1987. Linear Logic. *Theoret. Comput. Sci.* 50, 1 (1987), 1–102.

[35] Hannah Gommerstadt, Limin Jia, and Frank Pfenning. 2018. Session-Typed Concurrent Contracts. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 771–798. https://doi.org/10.1007/978-3-319-89884-1_27

[36] Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR'93*, Eike Best (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 509–523.

[37] Kohei Honda and Olivier Laurent. 2010. An exact correspondence between a typed pi-calculus and polarised proof-nets. *Theor. Comput. Sci.* 411, 22-24 (2010), 2223–2238.

[38] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, Chris Hankin (Ed.). Springer, 122–138.

[39] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luis Caires, et al. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1 (2016), 3.

[40] Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. 2020. Multiparty Session Programming With Global Protocol Combinators. In *34th European Conference on Object-Oriented Programming, ECOOP 2020 (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:30.

[41] Jules Jacobs and Stephanie Balzer. 2023. Higher-Order Leak and Deadlock Free Locks. *Proc. ACM Program. Lang.* 7, POPL (2023), 1027–1057.

[42] Steve Klabnik and Carol Nichols. 2021. The Rust Programming Language. (2021).

[43] Wen Kokke and Ornela Dardha. 2021. Deadlock-free session types in linear Haskell. In *Haskell 2021: Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, Jurriaan Hage (Ed.). ACM, 1–13.

[44] Wen Kokke, Fabrizio Montesi, and Marco Peressotti. 2019. Better late than never: a fully-abstract semantics for classical processes. *Proc. ACM Program. Lang.* 3, POPL (2019), 24:1–24:29.

[45] Jean-Louis Krivine. 2007. A call-by-name Lambda-calculus Machine. *High. Order Symb. Comput.* 20, 3 (2007), 199–207.

[46] Yves Lafont. 1988. The Linear Abstract Machine. *Theor. Comput. Sci.* 59 (1988), 157–180.

[47] Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. 2020. Implementing Multiparty Session Types in Rust. In *Coordination Models and Languages Coordination 2020 (Lecture Notes in Computer Science, Vol. 12134)*. Springer, 127–136.

[48] Peter J. Landin. 1964. The Mechanical Evaluation of Expressions. *The Computer Journal, Volume 6, Issue 4, January 1964* 6, 4 (1964), 308–320.

[49] Olivier Laurent. 1999. Polarized Proof-Nets: Proof-Nets for LC. In *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99 (LNCS, Vol. 1581)*, Jean-Yves Girard (Ed.). Springer, 213–227.

[50] Sam Lindley and J. Garrett Morris. 2016. Embedding session types in Haskell. In *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, Geoffrey Mainland (Ed.). ACM, 133–145.

[51] Luís M. B. Lopes, Fernando M. A. Silva, and Vasco Thudichum Vasconcelos. 1999. A Virtual Machine for a Process Calculus. In *Principles and Practice of Declarative Programming, International Conference PPDP'99 (Lecture Notes in Computer Science, Vol. 1702)*, Gopalan Nadathur (Ed.). Springer, 244–260.

[52] Robin Milner. 1992. Functions as Processes. *Math. Struct. Comput. Sci.* 2, 2 (1992), 119–141.

[53] Robin Milner. 1993. Elements of interaction: Turing award lecture. *Commun. ACM* 36, 1 (1993), 78–89.

[54] Robin Milner. 1999. *Communicating and mobile systems - the Pi-calculus.* Cambridge University Press.

[55] Guillaume Munch-Maccagnoni. 2009. Focalisation and Classical Realisability. In *Computer Science Logic, 23rd international Workshop, CSL 2009 (LNCS, Vol. 5771)*, Erich Grädel and Reinhard Kahle (Eds.). Springer, 409–423.

[56] Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. 2018. A session type provider: compile-time API generation of distributed protocols with refinements in F#. In *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*, Christophe Dubach and Jingling Xue (Eds.). ACM, 128–138.

[57] Luca Padovani. 2017. A simple library implementation of binary sessions. *J. Funct. Program.* 27 (2017), e4.

[58] Jorge A Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear logical relations and observational equivalences for session-based concurrency. *Information and Computation* 239 (2014), 254–302.

[59] Frank Pfenning. 1995. Structural Cut Elimination. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS '95).* IEEE Computer Society, USA, 156.

[60] Frank Pfenning and Dennis Griffith. 2015. Polarized Substructural Session Types. In *Proc. of FoSSaCS 2015 (LNCS, Vol. 9034).* Springer, 3–22.

[61] F. Pfenning and K. Pruiksma. 2023. Relating Message Passing and Shared Memory, Proof-Theoretically. In *Coordination Models and Languages - COORDINATION 2023 (LNCS, Vol. 13908)*, Sung-Shik Jongmans and A. Lopes (Eds.). Springer, 3–27.

[62] Benjamin C. Pierce and David N. Turner. 2000. Pict: a programming language based on the Pi-Calculus. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, Gordon D. Plotkin, Colin Stirling, and Mads Tofte (Eds.). The MIT Press, 455–494.

[63] Diogo Poças, Diana Costa, Andreia Mordido, and Vasco T. Vasconcelos. 2023. System F$^{\mu}$ ømega with Context-free Session Types. In *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023 (LNCS, Vol. 13990)*, Thomas Wies (Ed.). Springer, 392–420.

[64] Zesen Qian, GA Kavvos, and Lars Birkedal. 2021. Client-server sessions in linear logic. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–31.

[65] Pedro Rocha. 2022. *CLASS: A Logical Foundation for Typeful Programming with Shared State.* Ph. D. Dissertation. NOVA School of Science and Technology. http://hdl.handle.net/10362/146523

[66] Pedro Rocha and Luís Caires. 2021. Propositions-as-types and Shared State. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–30.

[67] Pedro Rocha and Luís Caires. 2021. Propositions-as-Types and Shared State (Artifact). (May 2021). https://doi.org/10.5281/zenodo.5037493

[68] Pedro Rocha and Luís Caires. 2023. Safe Session-Based Concurrency with Shared Linear State. In *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023 (LNCS, Vol. 13990)*, Thomas Wies (Ed.). Springer, 421–450.

[69] Pedro Rocha and Luís Caires. 2023. Safe Session-Based Concurrency with Shared Linear State (Artifact). (January 2023). https://doi.org/10.5281/zenodo.7506064

[70] Davide Sangiorgi and David Walker. 2001. *PI-Calculus: A Theory of Mobile Processes.* Cambridge University Press, USA.

[71] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. Hao. 2019. Safer smart contract programming with Scilla. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 185:1–185:30.

[72] Bernardo Toninho. 2015. *A Logical Foundation for Session-Based Concurrent Computation.* Ph. D. Dissertation. NOVA School of Science and Technology.

[73] Bernardo Toninho. 2015. *A Logical Foundation for Session-based Concurrent Computation.* Ph. D. Dissertation. Carnegie Mellon University and NOVA University of Lisbon.

[74] B. Toninho, L. Caires, and F. Pfenning. 2012. Functions as Session-Typed Processes. In *FoSSaCS'12 (LNCS, 7213).*

[75] Bernardo Toninho, Luis Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 350–369.

[76] Bernardo Toninho, Luís Caires, and Frank Pfenning. 2014. Corecursion and non-divergence in session-typed processes. In *International Symposium on Trustworthy Global Computing*. Springer, 159–175.

[77] Bernardo Toninho, Luís Caires, and Frank Pfenning. 2021. A Decade of Dependent Session Types. In *PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming*, Niccolò Veltri, Nick Benton, and Silvia Ghilezan (Eds.). ACM, 3:1–3:3.

[78] Bernardo Toninho and Nobuko Yoshida. 2021. On Polymorphic Sessions and Functions: A Tale of Two (Fully Abstract) Encodings. *ACM Trans. Program. Lang. Syst.* 43, 2, Article 7 (June 2021), 55 pages.

[79] David N. Turner. 1996. *The polymorphic Pi-calculus : theory and implementation.* Ph. D. Dissertation. University of Edinburgh, UK.

[80] Vasco Thudichum Vasconcelos. 2005. Lambda and pi calculi, CAM and SECD machines. *J. Funct. Program.* 15, 1 (2005), 101–127.

[81] Philip Wadler. 1990. Recursive types for free! *manuscript* (1990).

[82] Philip Wadler. 2012. Propositions as Sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming* (Copenhagen, Denmark) *(ICFP '12).* Association for Computing Machinery, New York, NY, USA, 273–286.

[83] Philip Wadler. 2014. Propositions as Sessions. *Journal of Functional Programming* 24, 2-3 (2014), 384–418.

[84] Philip Wadler. 2015. Propositions as Types. *Commun. ACM* 58, 12 (2015), 75–84.

[85] Max Willsey, Rokhini Prabhu, and Frank Pfenning. 2016. Design and Implementation of Concurrent C0. In *Proceedings Fourth International Workshop on Linearity, LINEARITY 2016 (EPTCS, Vol. 238)*, Iliano Cervesato and Maribel Fernández (Eds.). 73–82.

[86] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

10 APPENDIX

10.1 Proofs for Section 4.1: Preservation and Progress for CLLB.

Lemma 4.6 (Non-full). *For $P \neq \mathsf{0}$ the following rule is (1) admissible and (2) invertible in* CLLB*:*

$$\frac{P \vdash^{\mathsf{B}} \Delta_P, x{:}A; \Gamma \quad Q \vdash^{\mathsf{B}} \Delta_Q, y{:}B; \Gamma \quad \Gamma; \Delta_q \vdash q : \overline{B} \triangleright A \quad B \text{ negative}}{\mathsf{cut}\ \{P\ |\overline{x}{:}A\ [q]\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_q; \Gamma}$$

For inversion (2), the derivations for P and Q are sub-derivations of the conclusion.

Proof. **(Admissibility)** By induction on the size of the queue q.

■ (Case $q = \mathsf{nil}$) We have $A = \overline{B}$, so A is positive and we conclude by [TCutE].

■ (Case $q = q'@c$) By Lemma 4.5 (1), there are E, Δ_1, Δ_2 such that $\Delta_q = \Delta_1, \Delta_2$, $\Gamma; \Delta_1 \vdash q' : \overline{B} \triangleright E$ and $\Gamma; \Delta_2 \vdash c : E \triangleright A$. We consider each case for $\Gamma; \Delta_2 \vdash c : E \triangleright A$.

■ (Subcase $c = \checkmark$) We have $E = \mathbf{1}$ and $A = \emptyset$ and $\Delta_2 = \emptyset$ and $\Gamma; \Delta_1 \vdash q' : \overline{B} \triangleright \mathbf{1}$. Hence $P = \mathsf{0}$, contradiction.

■ (Subcase $c = \mathsf{clos}(z, R)$) We have $E = T \otimes A$ and $R \vdash \Delta_2, z : T; \Gamma$, and $\Gamma; \Delta_1 \vdash q' : \overline{B} \triangleright T \otimes A$.

By [T$\otimes$], we derive $\mathsf{send}\ x(z.R); P \vdash^{\mathsf{B}} \Delta_2, \Delta_P, x{:}T \otimes A; \Gamma$.

By i.h., we conclude $\mathsf{cut}\ \{\mathsf{send}\ x(z.R); P\ |\overline{x}{:}T \otimes A\ [q']\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_q; \Gamma$.

By [TCut$\otimes$], we have $\mathsf{cut}\ \{P\ |\overline{x}{:}A\ [q'@c]\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_q; \Gamma$ where $q'@c = q$.

■ (Subcase $c = \#\mathsf{l}$) We have $E = \oplus_{\ell \in L} E_\ell$ and $\Gamma; \Delta_1 \vdash q' : \overline{B} \triangleright \oplus_{\ell \in L} E_\ell$ and $\Delta_2 = \emptyset$. By [T$\oplus$], we derive $\#\mathsf{l}\ x; P \vdash^{\mathsf{B}} \Delta_P, x : \oplus_{\ell \in L} E_\ell; \Gamma$. By i.h., we conclude $\mathsf{cut}\ \{\#\mathsf{l}\ x; P\ |\overline{x} : \oplus_{\ell \in L} E_\ell\ [q']\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_q; \Gamma$.

By [TCut$\oplus$], $\mathsf{cut}\ \{P\ |\overline{x}{:}A\ [q'@c]\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_q; \Gamma$ where $q'@c = q$.

■ (Subcase $c = \mathsf{ty}(T)$) We have $E = \exists X.F$ and $\Gamma; \Delta_1 \vdash q' : \overline{B} \triangleright \exists X.F$ and $\Delta_2 = \emptyset$ and $A = \{T/X\}F$.

By [T$\exists$], we derive $\mathsf{sendty}\ x(T); P \vdash^{\mathsf{B}} \Delta_P, x : \exists X.F; \Gamma$.

By i.h., we conclude $\mathsf{cut}\ \{\mathsf{sendty}\ x(T); P\ |\overline{x} : \exists X.F\ [q']\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_q; \Gamma$.

By [TCut$\exists$], $\mathsf{cut}\ \{P\ |\overline{x}{:}\{T/X\}F\ [q'@c]\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_q; \Gamma$ where $q'@c = q$ and $\{T/X\}F = A$.

■ (Subcase $c = \mathsf{clos!}()$) We have $E = !C$ and $A = \emptyset$ and $\Delta_2 = \emptyset$ and $\Gamma; \Delta_1 \vdash q' : \overline{B} \triangleright !C$. Hence $P = \mathsf{0}$, contradiction.

■ (Subcase $c = \mathsf{step}$) We have $E = \mu X.F$ and $\Gamma; \Delta_1 \vdash q' : \overline{B} \triangleright \mu X.F$ and $\Delta_2 = \emptyset$ and $A = \{\mu X.F/X\}F$.

By [Tμ], we derive $\mathsf{unfold}_\mu\ x; P \vdash^{\mathsf{B}} \Delta_P, x : \mu X.F; \Gamma$.

By i.h., we conclude $\mathsf{cut}\ \{\mathsf{unfold}_\mu\ x; P\ |\overline{x} : \mu X.F\ [q']\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_q; \Gamma$.

By [TCutμ], $\mathsf{cut}\ \{P\ |\overline{x} : \{\mu X.F/X\}F\ [q'@c]\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_q; \Gamma$ where $q'@c = q$ and $\{\mu X.F/X\}F = A$.

(Inversion) By induction on the size of the queue q. In each case, we may verify that the proofs of the extracted premises are always subderivations of the conclusion.

■ (Case $q = \mathsf{nil}$) Then the conclusion is derived by [TCutE], so $A = \overline{B}$, $P \vdash^{\mathsf{B}} \Delta_P, x{:}A; \Gamma$ and $Q \vdash^{\mathsf{B}} \Delta_Q, y{:}B; \Gamma$ and A is positive. So we also have $\Gamma; \Delta_q \vdash q : \overline{B} \triangleright A$ where $\Delta_q = \emptyset$ and B is negative. Trivially, the proofs of premises are smaller than the conclusion.

■ (Case $q = q'@c$) We consider each case for c.

■ (Subcase $c = \checkmark$) If the conclusion is derived by [TCut$\mathbf{1}$], contradiction since $P = \mathsf{0}$, not applicable.

■ (Subcase $c = \mathsf{clos!}(z, P)$) If the conclusion is derived by [TCut!], contradiction since $P = \mathsf{0}$, not applicable.

■ (Subcase $c = \mathsf{clos}(z, R)$) We have $\mathsf{cut}\ \{P\ |\overline{x}{:}A\ [q'@\mathsf{clos}(z, R)]\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_q; \Gamma$ derived by [TCut$\otimes$] from $\mathsf{cut}\ \{\mathsf{send}\ x(z.R); P\ |\overline{x} : T \otimes A\ [q']\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_q; \Gamma$.

By i.h. inversion, we have $\mathsf{send}\ x(z.R); P \vdash^{\mathsf{B}} \Delta_R, \Delta_P, x{:}T \otimes A; \Gamma$, $Q \vdash^{\mathsf{B}} \Delta_Q, y{:}B; \Gamma$, and $\Gamma; \Delta_{q'} \vdash q' : \overline{B} \triangleright T \otimes A$ with $\Delta_q = \Delta'_q, \Delta_R$ and B negative. By [T$\otimes$] inversion we have $P \vdash^{\mathsf{B}} \Delta_P, x{:}A; \Gamma$ and $R \vdash^{\mathsf{B}} \Delta_R, z{:}T; \Gamma$.

Since $\Gamma; \Delta_R \vdash \mathsf{clos}(z, R) : T \otimes A \triangleright A$. by Lemma 4.5 (2), we conclude $\Gamma; \Delta_q \vdash q : \overline{B} \triangleright A$.

■ (Subcase $c = \mathsf{ty}(U)$) $\mathsf{cut}\ \{P\ |\overline{x}{:}A\ [q'@\mathsf{ty}(U)]\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_q; \Gamma$ derived by [TCut$\exists$] from $\mathsf{cut}\ \{\mathsf{sendty}\ x(X); P\ |\overline{x} : \exists X.F\ [q']\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_q; \Gamma$ where $A = \{T/X\}F$.

By i.h. inversion, we have $\mathsf{sendty}\ x(X); P \vdash^{\mathsf{B}} \Delta_P, x{:}\exists X.F; \Gamma$, $Q \vdash^{\mathsf{B}} \Delta_Q, y{:}B; \Gamma$, and $\Gamma; \Delta_{q'} \vdash q' : \overline{B} \triangleright \exists X.F$ with $\Delta_q = \Delta'_q$ and B negative. By [T$\exists$] inversion we have $P \vdash^{\mathsf{B}} \Delta_P, x{:}A; \Gamma$.

Since $\Gamma; \vdash \mathsf{ty}(T) : \exists X.F \triangleright \{T/X\}F = A$. by Lemma 4.5 (2), we conclude $\Gamma; \Delta_q \vdash q : \overline{B} \triangleright A$.

■ (Subcase $c = \mathsf{step}$) $\mathsf{cut}\ \{P\ |\overline{x}{:}A\ [q'@\mathsf{step}]\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_q; \Gamma$ derived by [TCutμ] from $\mathsf{cut}\ \{\mathsf{unfold}_\mu\ x; P\ |\overline{x} : \mu X.F\ [q']\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_Q, \Delta_q; \Gamma$ where $A = \{\mu X.F/X\}F$.

By i.h. inversion, we have $\mathsf{unfold}_\mu\ x; P \vdash^{\mathsf{B}} \Delta_P, x{:}\mu X.F; \Gamma$, $Q \vdash^{\mathsf{B}} \Delta_Q, y{:}B; \Gamma$, and $\Gamma; \Delta_{q'} \vdash q' : \overline{B} \triangleright \mu X.F$ with $\Delta_q = \Delta'_q$ and B negative. By [T$\exists$] inversion we have $P \vdash^{\mathsf{B}} \Delta_P, x{:}A; \Gamma$. Since $\Gamma; \vdash \mathsf{step} : \mu X.F \triangleright \{\mu X.F/X\}F = A$. by Lemma 4.5 (2), we conclude $\Gamma; \Delta_q \vdash q : \overline{B} \triangleright A$.

The proof for the remaining cases follow the same pattern. □

Lemma 4.7 (Full). *The following rule is (1) admissible and (2) invertible in* CLLB*:*

$$\frac{Q \vdash^{\mathsf{B}} \Delta_Q, y{:}B; \Gamma \quad \Gamma; \Delta_q \vdash q : \overline{B} \triangleright \emptyset \quad B \textit{ negative}}{\mathsf{cut}\ \{\mathsf{0}\ |\overline{x}{:}\emptyset\ [q]\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_Q, \Delta_q; \Gamma}$$

For (2), the derivation for Q is a sub-derivation of the conclusion. We also have $q = q'@\checkmark$ or $q = q'@\mathsf{clos!}(z, R)$ for some q'.

Proof. We consider the case $q = q'@\checkmark$. To verify admissibility, assume $Q \vdash^{\mathsf{B}} \Delta_Q, y{:}B; \Gamma$, and $\Gamma; \Delta_q \vdash q'@\checkmark : \overline{B} \triangleright \emptyset$ and B negative. By Lemma 4.5 (1), $\Gamma; \Delta_q \vdash q' : \overline{B} \triangleright \mathbf{1}$. By Lemma 4.6 (admissibility), we derive $\mathsf{cut}\ \{\mathsf{close}\ x\ |\overline{x} : \mathbf{1}\ [q']\ y : B|\ Q\} \vdash^{\mathsf{B}} \Delta_Q, \Delta_q; \Gamma$. By [TCut$\mathbf{1}$], we have $\mathsf{cut}\ \{\mathsf{0}\ |\overline{x} : \emptyset\ [q'@\checkmark]\ y : B|\ Q\} \vdash^{\mathsf{B}} \Delta_Q, \Delta_q; \Gamma$.

For inversion, assume $\mathsf{cut}\ \{\mathsf{0}\ |\overline{x}{:}\emptyset\ [q]\ y{:}B|\ Q\} \vdash^{\mathsf{B}} \Delta_Q, \Delta_q; \Gamma$. By inversion [TCut$\mathbf{1}$], we have $\mathsf{cut}\ \{\mathsf{close}\ x\ |\overline{x} : \mathbf{1}\ [q']\ y : B|\ Q\} \vdash^{\mathsf{B}} \Delta_Q, \Delta_q; \Gamma$. By Lemma 4.6 (inversion), we have $\mathsf{close}\ x \vdash^{\mathsf{B}} x{:}\mathbf{1}; \Gamma$, and $Q \vdash^{\mathsf{B}} \Delta_Q, y{:}B; \Gamma$, and $\Gamma; \Delta_q \vdash q : \overline{B} \triangleright \mathbf{1}$, and B negative. Since $\Gamma; \Delta_q \vdash q'@\checkmark : \overline{B} \triangleright \emptyset$ by Lemma 4.5 (2) we conclude.

The case $q = q'@\mathsf{clos!}(z, R)$ is similar. □

Type substitution principles are relevant for polymorphism and recursion in the proof of type preservation .

Lemma 10.1 (Type Substitution). *The following hold*

1. *(Instantiation) Let $P \vdash^{\mathsf{B}}_\eta \Delta; \Gamma$. Then $\{A/X\}P \vdash_{\{A/X\}\eta} \{A/X\}(\Delta; \Gamma)$.*
2. *(Unfolding) Let $\mathsf{rec}\ Y(z, \vec{w}); P\ [z, \vec{w}] \vdash^{\mathsf{B}}_\eta \Delta, z : \nu X.A; \Gamma$. Then, $\{\mathsf{rec}\ Y(z, \vec{w}); P/Y\}P \vdash^{\mathsf{B}}_\eta \Delta, z : \{\nu X.A/X\}A; \Gamma$.*

Proof. (1) Proof by induction on the type derivations for $P \vdash^{\mathsf{B}}_\eta \Delta; \Gamma$. (2) We first prove the more general property (A): Let $\mathsf{rec}\ Y(z, \vec{w}); P\ [z, \vec{w}] \vdash_\eta \Delta, z : \nu X.A; \Gamma$, $Q \vdash^{\mathsf{B}}_{\eta''} \Delta'; \Gamma'$, and let $\eta'' = \eta', Y(z, \vec{w}) \mapsto \Delta, z : X; \Gamma$ where η' is any mapping extending η for which Y, X are fresh. Then, $\{\mathsf{rec}\ Y(z, \vec{w}); P/Y\}Q \vdash^{\mathsf{B}}_{\eta'} \{\nu X.A/X\}(\Delta'; \Gamma')$. Then, (2) results by [Tcorec] inversion and considering $\eta' = \eta$ and using (A) on $Q \vdash^{\mathsf{B}} \Delta'; \Gamma' = P \vdash^{\mathsf{B}} \Delta, z : A; \Gamma$. □

Theorem 4.8 (Preservation). *Let $P \vdash^{\mathsf{B}} \Delta; \Gamma$. We have*

1. *If $P \equiv^{\mathsf{B}} Q$, then $Q \vdash^{\mathsf{B}} \Delta; \Gamma$.*
2. *If $P \rightarrow^{\mathsf{B}} Q$, then $Q \vdash^{\mathsf{B}} \Delta; \Gamma$.*

Proof. (1) We verify that conversion rules for structural congruence $\equiv^{\mathsf{B}}$ are type preserving. For any equation of $\equiv^{\mathsf{B}}$, we rely on Lemma 4.6 or Lemma 4.7, to factor out queues in cuts. We illustrate with one case.

■ (Case [BM]) Assume $\mathsf{cut}\ \{P\ |\overline{a} : A\ [q]\ b : B|\ (Q \mid\mid R)\} \vdash^{\mathsf{B}} \Delta_P, \Delta_q, \Delta_Q, \Delta_R; \Gamma$ where $b \in \mathsf{fn}(Q)$. By Lemma 4.6 (inversion), $P \vdash^{\mathsf{B}} a : A, \Delta_P; \Gamma$ and $Q \mid\mid R \vdash^{\mathsf{B}} b : B, \Delta_Q, \Delta_R; \Gamma$ and $\Gamma; \Delta_q \vdash q : \overline{B} \triangleright A$. So $Q \vdash^{\mathsf{B}} b : B, \Delta_Q; \Gamma$ and $R \vdash^{\mathsf{B}} \Delta_R; \Gamma$. By Lemma 4.6 (admissibility), $\mathsf{cut}\ \{P\ |\overline{a} : A\ [q]\ b : B|\ Q\} \vdash^{\mathsf{B}} \Delta_P, \Delta_q, \Delta_Q; \Gamma$. By [Tmix], $\mathsf{cut}\ \{P\ |a : A\ [q]\ b : B|\ Q\} \mid\mid R$.

(2) We verify that conversion rules for reduction $\xrightarrow{\mathsf{B}}$ are type preserving.

■ (Case [fwdp]) $P = \mathsf{cut}\ \{Q'\ |\overline{z}{:}A\ [q_1]\ x{:}\overline{B}|\ \mathsf{cut}\ \{\mathsf{fwd}\ x\ y\ |\overline{y}{:}B\ [q_2]\ w{:}C|\ P'\}\} \vdash^{\mathsf{B}} \Delta; \Gamma$.

If $q_2 = \mathsf{nil}$, and then $B = \overline{C}$, hence $Q = \mathsf{cut}\ \{Q'\ |\overline{z}{:}A\ [q_1]\ w{:}C|\ P'\} \vdash^{\mathsf{B}} \Delta; \Gamma$.

Otherwise $q_2 \neq \mathsf{nil}$. Let $F_2 = \mathsf{cut}\ \{\mathsf{fwd}\ x\ y\ |\overline{y}{:}B\ [q_2]\ w{:}C|\ P'\}$ where $F_2 \vdash^{\mathsf{B}} \Delta_2, x{:}\overline{B}$ and $\Delta = \Delta_1, \Delta_2$. By Lemma 4.6 (inversion), $\mathsf{fwd}\ x\ y \vdash^{\mathsf{B}} x : \overline{B}, y : B$, $P' \vdash w{:}C, \Delta_P; \Gamma$ and $\Delta_2 = \Delta_{q_2}, \Delta_P$ and $\Gamma; \Delta_{q_2} \vdash q_2 : \overline{C} \triangleright B$.

Let $F_1 = \mathsf{cut}\ \{Q'\ |\overline{z}{:}A\ [q_1]\ x{:}\overline{B}|\ F_2\}$ with $F_1 \vdash^{\mathsf{B}} \Delta_1; \Gamma$. By Lemma 4.6 (inversion), $Q' \vdash^{\mathsf{B}} \Delta_{Q'}, z : A; \Gamma$ and $\Gamma; \Delta_{q_1} \vdash q_1 : B \triangleright A$ and $\Delta_1 = \Delta_{q_1}, \Delta_{Q'}$. By Lemma 4.5 (2), $\Gamma; \Delta_{q_1}, \Delta_{q_2} \vdash q_2@q_1 : \overline{C} \triangleright A$. By Lemma 4.6 (admissibility) we conclude $\mathsf{cut}\ \{Q'\ |z{:}A\ [q_2@q_1]\ w{:}C|\ P'\} \vdash^{\mathsf{B}} \Delta; \Gamma$.

■ (Case of [$\otimes$]) Let $\mathsf{cut}\ \{\mathsf{send}\ x(z.P); Q\ |\overline{x} : A \otimes C\ [q]\ y : B|\ R\} \vdash^{\mathsf{B}} \Delta; \Gamma$. By Lemma 4.6 (inversion), we have $\Delta = \Delta_P, \Delta_R, \Delta_q, \Delta_Q$ and $\mathsf{send}\ x(z.P); Q \vdash^{\mathsf{B}} \Delta_P, x : A$ and $R \vdash \Delta_R, y : B$ and $\Gamma; \Delta_q \vdash q : \overline{B} \triangleright A \otimes C$.

By inversion [T$\otimes$], $\Delta_P = \Delta'_P, \Delta''_P$ and $Q \vdash^{\mathsf{B}} \Delta''_P, x : C; \Gamma$ and $P \vdash^{\mathsf{B}} \Delta'_P, z : A, \Gamma$. We have $\Gamma; \Delta'_P \vdash \mathsf{clos}(z, P) : A \otimes C \triangleright C$.

By Lemma 4.5 (2), $\Gamma; \Delta_q, \Delta'_P \vdash q@\mathsf{clos}(z, P) : \overline{B} \triangleright C$. By Lemma 4.6, $\mathsf{cut}\ \{Q\ |\overline{x} : C\ [q@\mathsf{clos}(z, P)]|\ y : B|\ R\} \vdash^{\mathsf{B}} \Delta; \Gamma$.

■ (Case [$\parr$]) Let $\mathsf{cut}\ \{P\ |\overline{x} : A\ [q]\ y : B|\ \mathsf{recv}\ y(w); Q\} \vdash^{\mathsf{B}} \Delta$, with $q = \mathsf{clos}(z, R)@q'$. We assume $P \neq \mathsf{0}$, the proof for case $P = \mathsf{0}$ is similar and simpler, using Lemma 4.7 (inversion).

By Lemma 4.6 (inversion), $P \vdash \Delta_P, x : A$ and $\mathsf{recv}\ y(w); Q \vdash^{\mathsf{B}} \Delta_Q, y : B$ and $\Gamma; \Delta_q \vdash \mathsf{clos}(z, R)@q' : \overline{B} \triangleright A$, where $\Delta = \Delta_P, \Delta_Q, \Delta_q$. By [T$\parr$] inversion we have $B = \overline{T} \parr \overline{C}$, for some T, C and $Q \vdash^{\mathsf{B}} w : \overline{T}, \Delta_Q, y : \overline{C}$ and $\overline{B} = T \otimes C$.

By Lemma 4.5 (1), there are $\Delta_{p_1}, \Delta_{p_2} = \Delta_q$ such that $\Gamma; \Delta_{p_1} \vdash \mathsf{clos}(z, R) : T \otimes C \triangleright C$ and $\Gamma; \Delta_{p_2} \vdash q' : C \triangleright A$, and $R \vdash z : T, \Delta_{p_1}; \Gamma$. If $q' = \mathsf{nil}$ we have $\mathsf{cut}\ \{P\ |\overline{x} : A\ [\mathsf{nil}]\ y : \overline{C}|\ (\mathsf{cut}\ \{R\ |\overline{z}_1 : T\ [\mathsf{nil}]\ w : \overline{T}|\ Q\})^{\mathsf{p}}\}^{\mathsf{r}} \vdash^{\mathsf{B}} \Delta_P, \Delta_q, \Delta_Q$. If $q' \neq \mathsf{nil}$, by [TCutE] we conclude $\mathsf{cut}\ \{R\ |\overline{z}_1 : T_1\ [\mathsf{nil}]\ w : \overline{T}_1|\ Q\}^{\mathsf{p}} \vdash^{\mathsf{B}} \Delta_{p_1}, \Delta_Q, y : \overline{C}; \Gamma$. By Lemma 4.6 (admissibility), we have $\mathsf{cut}\ \{P\ |\overline{x} : A\ [q']\ y : \overline{C}|\ (\mathsf{cut}\ \{R\ |\overline{z}_1 : T\ [\mathsf{nil}]\ w : \overline{T}|\ Q\})\} \vdash^{\mathsf{B}} \Delta_P, \Delta_q, \Delta_Q$.

■ (Case [$\exists$]) Let $\mathsf{cut}\ \{\mathsf{sendty}\ x(T); P\ |\overline{x} : \exists X.A\ [q]\ y : B|\ R\} \vdash^{\mathsf{B}} \Delta; \Gamma$. By Lemma 4.6 (inversion), we have $\Delta = \Delta_P, \Delta_R, \Delta_q$ and $\mathsf{sendty}\ x(T); P \vdash^{\mathsf{B}} \Delta_P, x : A$ and $R \vdash \Delta_R, y : B$ and $\Gamma; \Delta_q \vdash q : \overline{B} \triangleright \exists X.A$.

By inversion [T$\exists$], $P \vdash^{\mathsf{B}} \Delta_P, z : \{T/X\}A, \Gamma$. We also have $\Gamma; \vdash \mathsf{ty}(T) : \exists X.A \triangleright \{T/X\}A$.

By Lemma 4.5 (2), $\Gamma; \Delta_q \vdash q@\mathsf{ty}(T) : \overline{B} \triangleright \{T/X\}A$. By Lemma 4.6, $\mathsf{cut}\ \{P\ |\overline{x} : \{T/X\}A\ [q@\mathsf{ty}(T)]|\ y : B|\ R\} \vdash^{\mathsf{B}} \Delta; \Gamma$.

■ (Case [$\forall$]) Let $\mathsf{cut}\ \{P\ |\overline{x} : A\ [q]\ y : B|\ \mathsf{recvty}\ y(X); Q\} \vdash^{\mathsf{B}} \Delta; \Gamma$, with $q = \mathsf{ty}(T)@q'$. We assume $P \neq \mathsf{0}$, the case $P = \mathsf{0}$ is similar and simpler, using Lemma 4.7 (inversion). By Lemma 4.6 (inversion), $P \vdash \Delta_P, x : A$ and $\mathsf{recvty}\ y(X); Q \vdash^{\mathsf{B}} \Delta_Q, y : B$ and $\Gamma; \Delta_q \vdash \mathsf{ty}(T)@q' : \overline{B} \triangleright A$, where $\Delta = \Delta_P, \Delta_Q, \Delta_q$. By [T$\forall$] inversion we have $B = \forall X.\overline{C}$, for some C and $Q \vdash^{\mathsf{B}} \Delta_Q, y : \overline{C}$ and $\overline{B} = \exists X.C$. By Lemma 4.5 (1), $\Gamma; \vdash \mathsf{ty}(T) : \exists X.C \triangleright \{T/X\}C$ and $\Gamma; \Delta_q \vdash q' : \{T/X\}C \triangleright A$. If $q' \neq \mathsf{nil}$, then by Lemma 10.1 (1), we have $\{T/X\}Q \vdash^{\mathsf{B}} \Delta_Q, y : \{T/X\}\overline{C}$, so $Q \vdash^{\mathsf{B}} \Delta_Q, y : \overline{\{T/X\}C}$. By Lemma 4.6 (admissibility) we conclude $\mathsf{cut}\ \{P\ |\overline{x} : A\ [q']\ y : \overline{\{T/X\}C}|\ \{T/X\}Q\}^{\mathsf{r}} \vdash^{\mathsf{B}} \Delta; \Gamma$. If $q' = \mathsf{nil}$, we conclude $\mathsf{cut}\ \{P\ |\overline{x} : A\ [\mathsf{nil}]\ y : \overline{\{T/X\}C}|\ \{T/X\}Q\}^{\mathsf{r}} \vdash^{\mathsf{B}} \Delta; \Gamma$..

■ (Case [μ]) Let $\mathsf{cut}\ \{\mathsf{unfold}_\mu\ x; P\ |\overline{x} : \mu X.A\ [q]\ y : B|\ R\} \vdash^{\mathsf{B}} \Delta; \Gamma$. By Lemma 4.6 (inversion), we have $\Delta = \Delta_P, \Delta_q$ and $\mathsf{unfold}_\mu\ x; P \vdash^{\mathsf{B}} \Delta_P, x : A$ and $R \vdash \Delta_R, y : B$ and $\Gamma; \Delta_q \vdash q : \overline{B} \triangleright \mu X.A$. By inversion [T$\mu$], $P \vdash^{\mathsf{B}} \Delta_P, z : \{\mu X.A/X\}A, \Gamma$. We also have $\Gamma; \vdash \mathsf{step} : \mu X.A \triangleright \{\mu X.A/X\}A$. By Lemma 4.5 (2), $\Gamma; \Delta_q \vdash q@\mathsf{step} : \overline{B} \triangleright \{\mu X.A/X\}A$. By Lemma 4.6, $\mathsf{cut}\ \{P\ |\overline{x} : \{\mu X.A/X\}A\ [q@\mathsf{step}]|\ y : B|\ R\} \vdash^{\mathsf{B}} \Delta; \Gamma$.

■ (Case [ν]) Let $\mathsf{cut}\ \{P\ |\overline{x} : A\ [q]\ y : B|\ \mathsf{unfold}_\nu\ y; Q\} \vdash^{\mathsf{B}} \Delta; \Gamma$, with $q = \mathsf{step}@q'$. We assume $P \neq \mathsf{0}$, the case $P = \mathsf{0}$ is similar and simpler, using Lemma 4.7 (inversion). By Lemma 4.6 (inversion), $P \vdash \Delta_P, x : A; \Gamma$ and

$\mathsf{unfold}_\nu\ y;Q \vdash^{\mathsf{B}} \Delta_Q, y : B;\Gamma$ and $\Gamma;\Delta_q \vdash \mathsf{step}@q' : \overline{B} \triangleright A$, where $\Delta = \Delta_P, \Delta_Q, \Delta_q$. By [T$\nu$] inversion $B = \nu X.\overline{C}$ for some C and $Q \vdash^{\mathsf{B}} \Delta_Q, y : \{\nu X.\overline{C}/X\}\overline{C}$ and $\overline{B} = \mu X.C$. By Lemma 4.5 (2), $\Gamma;\vdash \mathsf{step} : \mu X.C \triangleright \{\mu X.C/X\}C$ and $\Gamma;\Delta_q \vdash q' : \{\mu X.C/X\}C \triangleright A$. Since $\overline{\{\mu X.C/X\}C} = \{\nu X.\overline{C}/X\}\overline{C}$, by Lemma 4.6 (admissibility) in the case $q \neq \mathsf{nil}$ we conclude $\mathsf{cut}\ \{P \mid \overline{x} : A\ [q']\ y : \{\nu X.\overline{C}/X\}\overline{C} \mid Q\} \vdash^{\mathsf{B}} \Delta;\Gamma$. The case $q' = \mathsf{nil}$ follows from [TCutE]. □

Lemma 10.2 (Barbs inversion). *Let $P \vdash^{\mathsf{B}} \Delta;\Gamma$ and $P \downarrow_x$.*

(1) *If $x \in \Gamma$ then $P \equiv \mathsf{call}\ x(y);Q$, otherwise $x : A \in \Delta$ and*
(2) *$P \equiv \mathsf{fwd}\ x\ y \mid * \mid Q$, or*
(3) *If $A = \mathbf{1}$ then $P \equiv \mathsf{close}\ x \mid * \mid R$.*
(4) *If $A = \bot$ then $P \equiv \mathsf{wait}\ x;Q \mid * \mid R$.*
(5) *If $A = B \otimes C$ then $P \equiv \mathsf{send}\ x(y.Q);R \mid * \mid S$.*
(6) *If $A = B \parr C$ then $P \equiv \mathsf{recv}\ x(y);Q \mid * \mid R$.*
(7) *If $A = \oplus_{\ell \in L} E_\ell$ then $P \equiv \#\mathsf{l}\ x;Q$.*
(8) *If $A = \&_{\ell \in L} E_\ell$ then $P \equiv \mathsf{case}\ x\ \{|\#\ell \in L{:}Q_\ell\} \mid * \mid R$.*
(9) *If $A =!B$ then $P \equiv \mathsf{!}x(y);Q \mid * \mid R$.*
(10) *If $A =?B$ then $P \equiv \mathsf{?}x;Q \mid * \mid R$.*
(11) *If $A = \exists X.C$ then $P \equiv \mathsf{sendty}\ x(T);Q \mid * \mid R$.*
(12) *If $A = \forall X.C$ then $P \equiv \mathsf{recvty}\ x(X);Q \mid * \mid R$.*
(13) *If $A = \mu X.C$ then $P \equiv \mathsf{unfold}_\mu\ x;Q \mid * \mid R$.*
(14) *If $A = \nu X.C$ then $P \equiv \mathsf{unfold}_\nu\ x;Q \mid * \mid R$ or $P \equiv \mathsf{rec}\ x(Y,\vec{z});Q\ [x,\vec{z}] \mid * \mid R$*

Proof. Induction on the typing derivation of $P \vdash \Delta;\Gamma$. □

Lemma 4.9 (Liveness). *Let $P \vdash^{\mathsf{B}} \Delta;\Gamma$. If P is live then either $P \downarrow_x$ or $P \rightarrow^{\mathsf{B}}$.*

Proof. By induction on the derivation for $P \vdash \Delta;\Gamma$, and case analysis on the last typing rule. The result is immediate for introductions and forwarder, since in all such cases $P \downarrow_x$. We then consider the case of the cut rules.

■ (Case of [TcutE]) we have $P = \mathsf{cut}\ \{P_1 \mid \overline{x} : A\ [\mathsf{nil}]\ y : \overline{A} \mid P_2\} \vdash^{\mathsf{B}} \Delta', \Delta;\Gamma$, derived from $P_1 \vdash^{\mathsf{B}} \Delta', x : A;\Gamma$ and $P_2 \vdash^{\mathsf{B}} \Delta, y : \overline{A};\Gamma$, where A is positive. By the i.h. we conclude that $P_1 \rightarrow$ or $P_1 \downarrow_{x_1}$. So if $P_1 \rightarrow$, then $P \rightarrow$. Otherwise, $P_1 \downarrow_{x_1}$. If $x_1 \neq x$ then $P \downarrow_{x_1}$ with $x_1 \in \Delta$.

Otherwise $x_1 = x$. Since A is positive, by Lemma 10.2 (2,3,5,7,9,11,13), either $P_1 \equiv \mathsf{fwd}\ x\ v \mid * \mid R$ (a), or we must have $P_1 \equiv a(x);Q$ where $a(x)$ is a positive action (b).

In case (a) if $v \notin \Delta$ the both x_2 and v are bound in P by cuts, and P reduces by [fwdp]. Otherwise $v \in \Delta$ and $P_2 \downarrow_v$ and $P \downarrow_v$. In case (b) we must have $P \rightarrow$ by one of [$\mathbf{1}$], [$\otimes$], [$\oplus$], [$\exists$], [!], or [μ], depending on the form of $a(x)$.

■ (Case of [TCut$\otimes$]) We have $P = \mathsf{cut}\ \{P_1 \mid \overline{x} : A\ [q]\ y : B \mid P_2\} \vdash^{\mathsf{B}} \Delta;\Gamma$ where $q = q'@\mathsf{clos}(z,R)$. By Lemma 4.6 (inversion), we have that $P_1 \vdash^{\mathsf{B}} x : A, \Delta_{P_1};\Gamma$, to which the i.h. applies. Hence $P_1 \rightarrow$ or $P_1 \downarrow_{x_1}$.

If $x \neq x_1$ then $P_1 \downarrow_{x_1}$ and $P \downarrow_{x_1}$. If $x = x_1$ and A is a positive type, by the same reasoning as above for P_1 in case [TCutE] we conclude that $P \downarrow_w$ for some w or $P \rightarrow$ by some positive action by P_1 writing into the queue. Otherwise, since the queue q is not empty and the process is well-typed, if P_2 can perform an action in y then it will reduce by reading the value at the queue. We detail this reasoning.

By i.h. $P_2 \rightarrow$ or $P_2 \downarrow_z$. If $z \neq y$ then $P \downarrow_z$. Otherwise, $P_2 \downarrow_y$. By Lemma 4.6 (inversion), B is a negative type. Then by Lemma 10.2 (2,4,6,8,10,12,14), either $P_2 \equiv \mathsf{fwd}\ y\ v \mid * \mid R$ (a), or $P_2 \equiv a(y);Q$ where $a(y)$ is a negative action (b).

For case (a), if v is free in P_2 the $P\downarrow_v$, otherwise $P\longrightarrow$ by [fwdp], since both y, v are bound by cuts.

In case (b) we must have $P\longrightarrow$ by one of $[\mathbf{1}]$, $[\parr]$, $[\&]$, $[\forall]$, $[?]$, or $[\nu]$, $[\nu\nu]$, depending on the form of $a(y)$.

■ (Case of [TCut$\oplus$], [TCut$\exists$], [TCutμ]) Similarly to the case [TCut$\otimes$] above, we show that either $P_1\downarrow_w$ with $w\neq x$ or $P_1\longrightarrow$ or $P_2\downarrow_w$ with $w\neq y$ or $P_2\longrightarrow$ and then $P\downarrow_w$ or $P\longrightarrow$. Otherwise $P_1\downarrow_x$ and $P_2\downarrow_y$. If A is positive then $P\longrightarrow$ by P_1 adding a value to the queue. If A is negative then $P\longrightarrow$ by P_2 reading a value from the queue.

■ (Case of [TCut$\mathbf{1}$]) Let $P=\mathsf{cut}\ \{0\ |\overline{x}:\emptyset\ [q]\ y:B|\ P_2\}\ \vdash^{\mathsf{B}}\Delta;\Gamma$ where $q=r@\checkmark$ derived from $\mathsf{cut}\ \{\mathsf{close}\ x\ |\overline{x}:\mathbf{1}\ [r]\ y:B|\ P_2\}\ \vdash^{\mathsf{B}}\Delta;\Gamma$. By i.h. $P_2\longrightarrow$ or $P_2\downarrow_z$. If $z\neq y$ then $P\downarrow_x$. Otherwise, $P_2\downarrow_y$. By Lemma 4.7, either $B=\overline{T}\parr C$ (a) or $B=\bot$ (b). If $P_2\equiv\mathsf{fwd}\ y\ v\ |*|\ Q'$, then as for [TCutE] we conclude that $P\longrightarrow$ or $P\downarrow_v$. For (a), by Lemma 10.2 (6), $P_2\equiv\mathsf{recv}\ y(w);Q'\ |*|\ Q''$, and $P\longrightarrow$ by $[\parr]$. For (b), by Lemma 10.2 (5), $P_2\equiv\mathsf{wait}\ y;Q'\ |*|\ Q''$, and $P\longrightarrow$ by $[\bot]$.

(Case of [TCut!]) Let $\mathsf{cut!}\ \{y.R\ |!x:A|\ Q\}\ \vdash^{\mathsf{B}}\Delta;\Gamma$ derived from $R\vdash^{\mathsf{B}} y:A;\Gamma$ and $Q\vdash^{\mathsf{B}}\Delta;\Gamma,x:\overline{A}$. By i.h. either $Q\longrightarrow$ or $Q\downarrow_z$. If $z\neq x$ then $P\longrightarrow$ or $P\downarrow_z$. If $z=x$ then by Lemma 10.2 $Q\equiv\mathsf{call}\ x(y);Q\ |*|\ R$ and $P\longrightarrow$ by [call]. □

10.2 Proofs of Section 4.2: Correspondence between CLL and CLLB.

We illustrate reduction commutations used in the proofs, those for pairs $\longrightarrow^{\mathsf{B}p}/\longrightarrow^{\mathsf{B}n}$ are checked similarly.

Lemma 10.3. *Commutation (pos-neg-promote):*

$$\begin{aligned}
&\mathsf{cut}\ \{\mathsf{send}\ x(z.R);P\ |\overline{x}\ [\mathsf{clos}(u,S)@q]\ y|\ \mathsf{recv}\ y(w);Q\}\longrightarrow^{\mathsf{B}p}\\
&\mathsf{cut}\ \{P\ |\overline{x}\ [\mathsf{clos}(u,S)@q@\mathsf{clos}(z,R)]\ y|\ \mathsf{recv}\ y(w);Q\}\longrightarrow^{\mathsf{B}n}\ \mathsf{cut}\ \{P\ |\overline{x}\ [q@\mathsf{clos}(z,R)]\ y|\ \mathsf{cut}\ \{S\ |u[]w|\ Q\}\}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{cut}\ \{\mathsf{send}\ x(z.R);P\ |\overline{x}\ [\mathsf{clos}(u,S)@q]\ y|\ \mathsf{recv}\ y(w);Q\}\longrightarrow^{\mathsf{B}n}\\
&\mathsf{cut}\ \{\mathsf{send}\ x(z.R);P\ |\overline{x}\ [q]\ y|\ \mathsf{cut}\ \{S\ |u[]w|\ Q\}\}\longrightarrow^{\mathsf{B}p}\ \mathsf{cut}\ \{P\ |\overline{x}\ [q@\mathsf{clos}(z,R)]\ y|\ \mathsf{cut}\ \{S\ |u[]w|\ Q\}\}
\end{aligned}$$

Lemma 10.4. *Commutation (pos-neg-seq-commute):*

$$\begin{aligned}
&E_1[\mathsf{cut}\ \{X\ |y\ [q_1]\ \overline{a}|\ \mathsf{send}\ a(z.R);\mathsf{recv}\ b(w);Q\}]\longrightarrow^{\mathsf{B}p}\\
&E_2[\mathsf{cut}\ \{Y\ |\overline{x}\ [\mathsf{clos}(u,S)@q_2]\ b|\ \mathsf{recv}\ b(w);Q\}\longrightarrow^{\mathsf{B}n}\ E_3[\mathsf{cut}\ \{Y\ |\overline{x}\ [q_2]\ b|\ \mathsf{cut}\ \{S\ |u[]w|\ Q\}\}]
\end{aligned}$$

$$\begin{aligned}
&E_1[\mathsf{cut}\ \{X\ |y\ [q_1]\ \overline{a}|\ \mathsf{send}\ a(z.R);\mathsf{recv}\ b(w);Q\}]\equiv\\
&F_2[\mathsf{cut}\ \{W\ |\overline{x}\ [\mathsf{clos}(u,S)@q_2]\ b|\ \mathsf{recv}\ b(w);\mathsf{send}\ a(z.R);Q\}]\longrightarrow^{\mathsf{B}n}\\
&F_3[\mathsf{cut}\ \{Z\ |y\ [q_1]\ \overline{a}|\ \mathsf{cut}\ \{S\ |u[]w|\ \mathsf{send}\ a(z.R);Q\}\}\longrightarrow^{\mathsf{B}p}\ E_3[\mathsf{cut}\ \{Y\ |\overline{x}\ [q_2]\ b|\ \mathsf{cut}\ \{S\ |u[]w|\ Q\}\}]
\end{aligned}$$

Lemma 10.5. *Commutation (fwd-neg-comm):*

$$\begin{aligned}
&E[\mathsf{cut}\ \{P\ |\overline{z}\ [q_1]\ x|\ \mathsf{fwd}\ x\ y\ |\overline{y}\ [\mathsf{clos}(u,S)@q_2]\ b|\ \mathsf{recv}\ b(w);Q\}]\longrightarrow^{\mathsf{B}a}\\
&E[\mathsf{cut}\ \{P\ |\overline{z}\ [\mathsf{clos}(u,S)@q_2@q_1]\ b|\ \mathsf{recv}\ b(w);Q\}]\longrightarrow^{\mathsf{B}n}\ E[\mathsf{cut}\ \{P\ |\overline{z}\ [q_2@q_1]\ b|\ \mathsf{cut}\ \{S\ |u[]w|\ Q\}\}]
\end{aligned}$$

$$\begin{aligned}
&E[\mathsf{cut}\ \{P\ |\overline{z}\ [q_1]\ x|\ \mathsf{fwd}\ x\ y\ |\overline{y}\ [\mathsf{clos}(u,S)@q_2]\ b|\ \mathsf{recv}\ b(w);Q\}]\longrightarrow^{\mathsf{B}n}\\
&E[\mathsf{cut}\ \{P\ |\overline{z}\ [q_1]\ x|\ \mathsf{fwd}\ x\ y\ |\overline{y}\ [q_2]\ b|\ \mathsf{cut}\ \{S\ |u[]w|\ Q\}\}]\longrightarrow^{\mathsf{B}a}\ E[\mathsf{cut}\ \{P\ |\overline{z}\ [q_2@q_1]\ b|\ \mathsf{cut}\ \{S\ |u[]w|\ Q\}\}]
\end{aligned}$$

We recall the following notations.

(1) Write $P\longrightarrow^{\mathsf{B}p}Q$ for $P\longrightarrow^{\mathsf{B}}Q$ if this reduction is positive (uses $[\mathbf{1}]$, $[\otimes]$, $[\oplus]$, $[!]$, $[\exists]$, or $[\mu]$).

(2) Write $P\longrightarrow^{\mathsf{B}n}Q$ for $P\longrightarrow^{\mathsf{B}}Q$ if this reduction is negative or [call] (uses $[\bot]$, $[\parr]$, $[\&]$, $[?]$, [call], $[\forall]$, $[\nu]$ or $[\nu\mu]$).

(3) Write $P\longrightarrow^{\mathsf{B}a}Q$ for $P\longrightarrow^{\mathsf{B}}Q$ if this reduction is by [fwdp].

(4) Write $P\overset{\epsilon}{\longrightarrow}{}^{\mathsf{B}a}Q$ for $P\longrightarrow^{\mathsf{B}a}Q$ if this [fwdp] reduction acts on empty cuts.

(5) Write $P\longrightarrow^{\mathsf{Bap}}Q$ for $P\longrightarrow^{\mathsf{B}}Q$ if this reduction is positive or a forwarder.

(6) Write $P\longrightarrow^{\mathsf{Br}}Q$ for a positive action on a buffered cut with empty queue imediately followed by a matching negative action on the very same cut.

Lemma 4.13 (Commutations). *The following commutation properties of reductions hold.*

1. *Let* $P_1 \longrightarrow^{\mathsf{B}p} S \longrightarrow^{\mathsf{B}n} P_2$. *Then either (a)* $P_1 \longrightarrow^{\mathsf{Br}} P_2$, *or (b)* $P_1 \longrightarrow^{\mathsf{B}n} S' \longrightarrow^{\mathsf{B}p} P_2$ *for some* S'.
2. *Let* $P_1 \longrightarrow^{\mathsf{B}a} S \longrightarrow^{\mathsf{B}n} P_2$. *Then* $P_1 \longrightarrow^{\mathsf{B}n} S' \longrightarrow^{\mathsf{B}a} P_2$ *for some* S'.
3. *Let* $P_1 \longrightarrow^{\mathsf{Bap}} S \longrightarrow^{\mathsf{B}n} P_2$. *Then either (a)* $P_1 \longrightarrow^{\mathsf{Br}} P_2$, *or (b)* $P_1 \longrightarrow^{\mathsf{B}n} S' \longrightarrow^{\mathsf{Bap}} P_2$ *for some* S'.
4. *Let* $P_1 \longrightarrow^{\mathsf{Bap}} N \overset{\epsilon}{\Rightarrow}^{\mathsf{B}a} S \longrightarrow^{\mathsf{Br}} P_2$. *Then either (a)* $P_1 \overset{\epsilon}{\longrightarrow}^{\mathsf{B}a} N$ *or (b) there is* S' *such that* $P_1 \longrightarrow^{\mathsf{Br}} S' \Rightarrow^{\mathsf{Bap}} P_2$.

Proof. (1) Either (a) the reductions are in the same cut, or (b) the reductions are in different cuts. For (a), if reductions match we conclude. Otherwise they commute (by Lemma 10.3) so $P_1 \longrightarrow^{\mathsf{B}n} S \longrightarrow^{\mathsf{B}p} P_2$ for some S. For (b), if reductions are independent (different threads), commute, and we conclude. If the reductions are dependent (same thread), they commute (by Lemma 10.4) and we conclude.

(2) We consider two cases: either (a) the reductions are in the same cut, or (b) the reductions are in different cuts. For (a), the reductions commute (by Lemma 10.5), so we have $P \longrightarrow^{\mathsf{B}n} S \longrightarrow^{\mathsf{B}a} Q$ for some S. For (b), the reductions are independent and commute.

(3) By (1) and (2).

(4) Assume $P_1 \longrightarrow^{\mathsf{B}p} N$. Since the reductions in $S \longrightarrow^{\mathsf{Br}} P_2$ must act on the same cut with an initially empty queue, and all reductions $N \overset{\epsilon}{\Rightarrow}^{\mathsf{B}a} S$ are on empty cuts, the reduction $P_1 \longrightarrow^{\mathsf{B}p} N$ must act on a different cut of all those involved and thus commutes with $N \overset{\epsilon}{\Rightarrow}^{\mathsf{B}a} S \longrightarrow^{\mathsf{Br}} P_2$, and we conclude (b). If $P_1 \longrightarrow^{\mathsf{B}a} S$ then either this redex generates one empty cut, so $P_1 \overset{\epsilon}{\longrightarrow}^{\mathsf{B}a} S$, and we conclude (a), or, as in case (b), it must be independent of $N \overset{\epsilon}{\Rightarrow}^{\mathsf{B}a} S \longrightarrow^{\mathsf{Br}} P_2$. □

Lemma 4.11 (Simulation of CLL by CLLB). *Let* $P \vdash \emptyset;\emptyset$. *If* $P \rightarrow Q$ *then* $P^\dagger \Rightarrow^{\mathsf{B}} Q^\dagger$.

Proof. Each cut reduction of CLL is simulated by two reduction steps of CLLB in sequence: [⊗⅋] by [⊗] followed by [⅋]; [**1**⊥] by [**1**] followed by [⊥]; [!?] by [!] followed by [?]. In a closed CLL process a [fwd] reduction has the form $\mathsf{cut}\ \{Q\ |x : A|\ \mathsf{fwd}\ x\ y\ |y : A|\ P\} \rightarrow \mathsf{cut}\ \{Q\ |x : A|\ \{x/y\}P\}$, which, for $A+$, reduces by [fwdB] as $\mathsf{cut}\ \{Q\ |\overline{x}{:}A\ [\mathsf{nil}]\ z{:}\overline{A}|\ \mathsf{fwd}\ z\ w\ |\overline{w}\ [\mathsf{nil}]\ y|\ P\} \rightarrow_a \mathsf{cut}\ \{Q\ |\overline{x}{:}A\ [\mathsf{nil}]\ y{:}\overline{A}|\ P\}$. □

Lemma 4.14 (Postponing). *Let* $P \vdash^{\mathsf{B}} \emptyset;\emptyset$. *If* $P \Rightarrow^{\mathsf{Bap}} \longrightarrow^{\mathsf{B}n} Q$ *then either*

1. $P \longrightarrow^{\mathsf{B}n} R$ *and* $R \Rightarrow^{\mathsf{Bap}} Q$ *for some* R, *or;*
2. $P \overset{\epsilon}{\Rightarrow}^{\mathsf{B}a} \longrightarrow^{\mathsf{Br}} R$ *and* $R \Rightarrow^{\mathsf{Bap}} Q$ *for some* R.

Proof. By induction on $P(\longrightarrow^{\mathsf{Bap}})^* P'$.

(Base) We have $P \equiv P' \longrightarrow^{\mathsf{B}n} Q$, hence (1).

(Inductive) Case $P \longrightarrow^{\mathsf{Bap}} P' \Rightarrow^{\mathsf{Bap}} \longrightarrow^{\mathsf{B}n} Q$. Let (r0) $P \longrightarrow^{\mathsf{Bap}} P'$.

By i.h. for P' there is R' so that (c1) $P' \longrightarrow^{\mathsf{B}n} R'$ and $R' \Rightarrow^{\mathsf{Bap}} Q$, or (c2) $P' \overset{\epsilon}{\Rightarrow}^{\mathsf{B}a} \longrightarrow^{\mathsf{Br}} R'$ and $R' \Rightarrow^{\mathsf{Bap}} Q$.

Case (c1). We have $P \longrightarrow^{\mathsf{Bap}} P' \longrightarrow^{\mathsf{B}n} R'$ and $R' \Rightarrow^{\mathsf{Bap}} Q$. By Lemma 4.13 (3) on $P \longrightarrow^{\mathsf{Bap}} P' \longrightarrow^{\mathsf{B}n} R'$, either $P \longrightarrow^{\mathsf{Br}} R'$ and we conclude (2) ($R = R'$) or there is R such that $P \longrightarrow^{\mathsf{B}n} R \longrightarrow^{\mathsf{Bap}} R'$ and we conclude (1).

Case (c2). We have $P \longrightarrow^{\mathsf{Bap}} P' \overset{\epsilon}{\Rightarrow}^{\mathsf{B}a} \longrightarrow^{\mathsf{Br}} R'$ and $R' \Rightarrow^{\mathsf{Bap}} Q$. By Lemma 4.13 (4) either (a) $P \overset{\epsilon}{\longrightarrow}^{\mathsf{B}a} P'$ or (b) $P \longrightarrow^{\mathsf{Br}} R'' \longrightarrow^{\mathsf{Bap}} R'$. In case (a) $P \overset{\epsilon}{\Rightarrow}^{\mathsf{B}a} \longrightarrow^{\mathsf{Br}} R'$ we conclude (2) ($R = R'$), in case (b) we get (2) as well (with $P \overset{\epsilon}{\Rightarrow}^{\mathsf{B}a}$ empty). □

Theorem 4.15 (Operational correspondence CLL-CLLB). *Let* $P \vdash_{\mathsf{CLL}} \emptyset;\emptyset$.

1. *If* $P \Rightarrow R$ *then* $P^\dagger \Rightarrow^{\mathsf{B}} R^\dagger$.
2. *If* $P^\dagger \Rightarrow^{\mathsf{B}} Q$ *then there is* R *such that* $P \Rightarrow R$ *and* $R^\dagger \Rightarrow^{\mathsf{Bap}} Q$.

Proof. (1) Iterating Lemma 4.11.

(2) We first prove (A): if $P^{\dagger} \Rightarrow^{\mathsf{Bap}} \rightarrow^{\mathsf{B}n} Q$ then there is R such that $P \Rightarrow R$ and $R^{\dagger} \Rightarrow^{\mathsf{Bap}} Q$. Assume $P^{\dagger} \Rightarrow^{\mathsf{Bap}} \rightarrow^{\mathsf{B}n} Q$. By Lemma 4.14 (2) we have $P^{\dagger} \overset{\epsilon}{\Rightarrow}^{\mathsf{B}a} \rightarrow^{\mathsf{Br}} R'$ and $R' \Rightarrow^{\mathsf{Bap}} Q$ for some R' ((1) cannot apply, since in $P^{\dagger}$ all queues are empty). Then there is P^* such that $P \Rightarrow P^*$ by [fwd] and $P^* \rightarrow R'$, where $P^{*\dagger} = R'$. So $P \Rightarrow R'$ and $R' = R^{\dagger}$ for some R since all cuts are empty in R'. Since $R^{\dagger} \Rightarrow^{\mathsf{Bap}} Q$ we conclude (A). Now, we note that $P^{\dagger} \Rightarrow^{\mathsf{B}} Q$ implies $P^{\dagger} (\Rightarrow^{\mathsf{Bap}} \rightarrow^{\mathsf{B}n})^* \Rightarrow^{\mathsf{Bap}} Q$. We then iterate (A) on each $\Rightarrow^{\mathsf{Bap}} \rightarrow^{\mathsf{B}n}$ reduction sub-sequence to conclude (2). □

10.3 Proofs of Section 5: The Linear SAM and its Correctness

In the proof of Lemma 5.5 we map each SAM transition into a reduction of the CLLB process P encoded by the machine configuration. Lemma 10.6 in useful to identify the action redex $\mathcal{A}$ via a context decomposition $P \equiv^{\mathsf{B}} E[F[\mathcal{A}]]$.

Lemma 10.6 (Encoding to Context). *Let $P \vdash_{\mathsf{B}} \emptyset$ and $P \overset{\mathsf{enc}*}{\Mapsto} (Q, H)$.*
Then $P \equiv_{\mathsf{B}} E[Q]$, where $E[\square] = F_{z_1}[F_{z_2}[\ldots[F_{z_n}[\square]]\ldots]]$, $H = S_{x_1 y_1} \cdots S_{x_n y_n}$, and
for every i, F_{z_i} is a one-hole cut context binding z_i and $S_{x_i y_i}$ a session record such that either:
(a) $z_i = x_i$ and $F_{z_i} = \mathsf{cut}\ \{\square\ |\overline{x}_i{:}A\ [q]\ y_i{:}B|\ R\}$ and $S_{x_i y_i} = \overline{x}_i{:}A\langle q, R(y_i)\rangle y_i{:}B$ in write-mode, or
(b) $z_i = y_i$ and $F_{z_i} = \mathsf{cut}\ \{R\ |\overline{x}_i{:}A\ [q]\ y_i{:}B|\ \square\}$ and $S_{x_i y_i} = \overline{x}_i{:}A\langle q, R(x_i)\rangle y_i{:}B$ in read-mode.

Proof. By induction on $P \overset{\mathsf{enc}*}{\Mapsto} (Q, H)$, we consider at each step one of the two cases of Definition 5.3. □

Lemma 5.5 (SAM-CLLB Step Safety). *Let $P \vdash^{\mathsf{B}} \emptyset$ and $enc(P)$ a ready SAM configuration. If $enc(P)$ is live then (1) there is C ready such that $enc(P) \Mapsto C$ and (2) there is Q such that $P \rightarrow^{\mathsf{B}} Q$ and $Q \overset{\mathsf{enc}*}{\Mapsto} C \overset{\mathsf{cut}*}{\Mapsto} enc(Q)$.*

Proof. Assume $enc(P) = (\mathcal{A}, H)$ live. We consider each process construct $\mathcal{A}$.

■ (Case of $\mathcal{A} = \mathsf{fwd}\ x\ y$) Wlog, assume that $x{:}\overline{B}$ is negative, so $y{:}B$ is positive. By Lemma 10.6, there are contexts E, F_x, F_y such that $P \equiv^{\mathsf{B}} E[F_x[F_y[\mathsf{fwd}\ x\ y]]]$. Hence, x and y must be endpoints of different session records, arising from the cuts of F_x and F_y, where $P \equiv^{\mathsf{B}} E[\mathsf{cut}\ \{P_1\ |\overline{z}{:}A\ [q_1]\ x{:}\overline{B}]|\ \mathsf{cut}\ \{\mathsf{fwd}\ x\ y\ |\overline{y}{:}B\ [q_2]\ w{:}C]|\ P_2\}\}]$ and $H = H'[z\langle q_1, U(z)\rangle x{:}\overline{B}][y{:}B\langle q_2, V(w)\rangle w]$ with $y \langle - \rangle w$ in write-mode (B+ and $\mathsf{step} \notin q_2$) and $z \langle - \rangle x$ in read-mode (q_1 step-terminated, or not A+ and $\mathsf{step} \notin q_1$). Thus we have (1) $enc(P) = (\mathsf{fwd}\ x\ y, H'[z{:}A\langle q_1, P_1\rangle x{:}\overline{B}][y{:}B\langle q_2, P_2\rangle w{:}C]) \Mapsto C$ by [Sfwd] where $C = (P_2, H'[z{:}A\langle q_2@q_1, P_1\rangle w{:}C])$. C is ready with $z \langle - \rangle w$ in read-mode. Then (2) $P \rightarrow^{\mathsf{B}} Q \equiv E[\mathsf{cut}\ \{P_1\ |\overline{z}{:}A\ [q_2@q_1]\ w{:}C]|\ P_2\}]$ by [fwd] , and $Q \overset{\mathsf{enc}*}{\Mapsto} C$ where $C \overset{\mathsf{cut}*}{\Mapsto} enc(Q)$ by Lemma 5.4 (2).

■ (Case of $\mathcal{A} = \mathsf{close}\ x$) By Lemma 10.6, there are contexts G, F_x such that $P \equiv^{\mathsf{B}} E[F_x[\mathsf{close}\ x]]$ and $P \equiv^{\mathsf{B}} E[\mathsf{cut}\ \{\mathsf{close}\ x\ |\overline{x}{:}\mathbf{1}\ [q]\ y{:}B]|\ R(y)\}]$, with $H = H'[x{:}\mathbf{1}\langle q, R(y)\rangle y{:}B]$ and $x \langle - \rangle y$ in write-mode.

Then $enc(P) \Mapsto (R(y), H'[x{:}\emptyset\langle q@\checkmark, 0\rangle y{:}B]) = C$ by [S$\mathbf{1}$], and C is ready with $x \langle - \rangle y$ in read-mode. Then (2) $P \rightarrow^{\mathsf{B}} Q \equiv E[\mathsf{cut}\ \{0\ |\overline{x}{:}\emptyset\ [q@\checkmark]\ y{:}B]|\ R(y)\}]$ by [$\mathbf{1}$] and $Q \overset{\mathsf{enc}*}{\Mapsto} C$ where $C \overset{\mathsf{cut}*}{\Mapsto} enc(Q)$ by Lemma 5.4 (2).

■ (Case of $\mathcal{A} = \mathsf{wait}\ y; R$) By Lemma 10.6, there are contexts E, F_y such that $P^{\mathsf{B}} \equiv E[F_y[\mathsf{wait}\ y; R]]$ and $P \equiv^{\mathsf{B}} E[\mathsf{cut}\ \{P\ |\overline{x}{:}A\ [q]\ y{:}\bot]|\ \mathsf{wait}\ y; R\}]$ and $H = H'[x{:}A\langle q, P\rangle y{:}\bot]$ with the session record in read-mode. Since $\Delta \vdash q : \mathbf{1} \triangleright A$, we must have $q = \checkmark$ and $A = \emptyset$ and $P = 0$. Thus, $H = H'[x{:}\emptyset\langle q, 0\rangle y{:}\bot]$. Then (1) $enc(P) \Mapsto C = (R, H')$, with C ready. Hence (2) $P \rightarrow^{\mathsf{B}} Q = E[R]$. We have $Q \overset{\mathsf{enc}*}{\Mapsto} C \overset{\mathsf{cut}*}{\Mapsto} enc(Q)$ as above.

■ (Case of $\mathcal{A} = \mathsf{send}\ x(z.R(z)); U(x)$) By Lemma 10.6, for E, F_x we have $P \equiv^{\mathsf{B}} E[F_x[\mathsf{send}\ x(z.R(z)); U(x)]] \equiv^{\mathsf{B}} E[\mathsf{cut}\ \{\mathsf{send}\ x(z.R(z)); U(x)\ |\overline{x}{:}T \otimes A\ [q]\ y{:}B]|\ S(y)\}]$ and $H = H'[x{:}T \otimes A\langle q, S(y)\rangle y{:}B]$ with $x \langle - \rangle y$ in write-mode. Then, if A+ (a), $enc(P) \Mapsto C_1 = (U(x), H'[x{:}A\langle q@\mathsf{clos}(z, R(z)), S(y)\rangle y{:}B])$ by [S$\otimes$]; if A− (b), $enc(P) \Mapsto C_2 = (S(y), H'[x{:}A\langle q@\mathsf{clos}(z, R(z)), U(x)\rangle y{:}B])$ by [S$\otimes$]. Notice that C_1 is ready ($x \langle - \rangle y$ in write-mode), C_2 is ready ($x \langle - \rangle y$

in read-mode). For (a), $P \rightarrow^{\mathsf{B}} Q = E[\mathsf{cut}\ \{U(x)\ |\overline{x}{:}A{+}\ [q@\mathsf{clos}(z, R(z))]\ y{:}B]|\ S(y)\}]$ by $[\otimes]$, and $Q \overset{\mathsf{enc}*}{\Mapsto} C_1 \overset{\mathsf{cut}*}{\Mapsto} enc(Q)$; for (b), $P \rightarrow^{\mathsf{B}} Q = E[\mathsf{cut}\ \{U(x)\ |\overline{x}{:}A{-}\ [q@\mathsf{clos}(z, R(z))]\ y{:}B]|\ S(y)\}]$ by $[\otimes]$, and $Q \overset{\mathsf{enc}*}{\Mapsto} C_2 \overset{\mathsf{cut}*}{\Mapsto} enc(Q)$.

■ (Case of $\mathcal{A} = \mathsf{recv}\ y(w{:}\overline{A}); U(y, w)$) By Lemma 10.6, $P \equiv^{\mathsf{B}} E[F_y[\mathsf{recv}\ y(w{:}\overline{A}); U(y, w)]] \equiv^{\mathsf{B}}$ $E[\mathsf{cut}\ \{S(x)\ |\overline{x}{:}C\ [q]\ y{:}\overline{A} \parr D]|\ \mathsf{recv}\ y(w{:}\overline{A}); U(y, w)\}]$, for some contexts E, F_y. Thus, $H = H'[x{:}C\langle q, S(x)\rangle y{:}\overline{A} \parr D]$ with $x \langle - \rangle y$ in read-mode. Since $\Delta \vdash q : A \otimes \overline{D} \triangleright C$, we have $q = \mathsf{clos}(z{:}A, R(z))@q'$ ($\Delta \vdash \mathsf{nil} : A \otimes \overline{D} \triangleright C$ is not derivable, because then not $+C$ would hold by the read-mode condition). We prove the statement for $A+$, the case $A-$ is similar.

We have (1) $enc(P) \Mapsto C$ where either (a) $C = C_1 = (R(z), H'[x{:}C\langle q', S(x)\rangle y{:}D][z{:}A\langle \mathsf{nil}, U(y, w)\rangle w{:}\overline{A}])$ (if $q' \neq \mathsf{nil}$), or (b) $C = C_2 = (R(z), H'[y{:}D\langle \mathsf{nil}, S(x)\rangle x{:}C][z{:}A\langle \mathsf{nil}, U(y, w)\rangle w{:}\overline{A}])$ (if $q' = \mathsf{nil}$). Notice that in (a) C_1 is ready ($x \langle - \rangle y$ in read-mode and $z \langle - \rangle w$ in write-mode); in (b) C_2 is ready, ($y \langle - \rangle x$ in write-mode and $z \langle - \rangle w$ in write-mode). For (2), in (a) then $P \rightarrow^{\mathsf{B}} Q = E[\ \mathsf{cut}\ \{S(x)\ |\overline{x}{:}C\ [q]\ y{:}D|\ \mathsf{cut}\ \{R(z)\ |\overline{z}{:}A\ [\mathsf{nil}]\ w{:}\overline{A}|\ U(y, w)\}\}]$ by $[\parr]$, and $Q \overset{\mathsf{enc}*}{\Mapsto} \overset{\mathsf{enc}}{\Mapsto} C_1 \overset{\mathsf{cut}*}{\Mapsto} enq(Q)$; in (b) then $P \rightarrow^{\mathsf{B}} Q = E[\ \mathsf{cut}\ \{S(x)\ |x{:}C\ [\mathsf{nil}]\ \overline{y}{:}D|\ \mathsf{cut}\ \{R(z)\ |\overline{z}{:}A\ [\mathsf{nil}]\ w{:}\overline{A}|\ U(y, w)\}\}]$ We have $Q \overset{\mathsf{enc}*}{\Mapsto} \overset{\mathsf{enc}}{\Mapsto} C_2 \overset{\mathsf{cut}*}{\Mapsto} enq(Q)$.

■ (Case of $\mathcal{A} = \#\mathsf{l}\ x; R(x)$) By Lemma 10.6, $P \equiv^{\mathsf{B}} E[\#\mathsf{l}\ x; R(x)]\ \equiv^{\mathsf{B}} E[F_x[\mathsf{cut}\ \{\#\mathsf{l}\ x; R(x)\ |\overline{x}{:}\oplus_{\ell \in L} A_\ell\ [q]\ y{:}B]|\ S(y)\}]]$ for E, F_x, and $H = H'[x{:}\oplus_{\ell \in L} A_\ell \langle q, S(y)\rangle y{:}B]$, with $x \langle - \rangle y$ in write-mode. So (1) $enc(P) \Mapsto C$ where (a) $C = C_1 = (R(x), H'[x{:}A_{\#\mathsf{l}}\langle q@\#\mathsf{l}, S(y)\rangle y{:}B])$ if $A_{\#\mathsf{l}}+$ and $C = C_2 = (S(y), H'[x{:}A_{\#\mathsf{l}}\langle q@\#\mathsf{l}, R(x)\rangle y{:}B])$ if $A_{\#\mathsf{l}}-$, with C_i ready in (a) or (b). For (2), let $P \rightarrow^{\mathsf{B}} Q = E[\mathsf{cut}\ \{R(x)\ |\overline{x}{:}A_{\#\mathsf{l}}\ [q@\#\mathsf{l}]\ y{:}B]|\ S(y)\}]$ by $[\oplus]$, and $Q \overset{\mathsf{enc}*}{\Mapsto} C_i \overset{\mathsf{cut}*}{\Mapsto} enq(Q)$ in (a) or (b).

■ (Case of $\mathcal{A} = \mathsf{case}\ y\ \{|\#\ell \in L{:}P_\ell(y)\}$) Similar to $\mathcal{A} = \mathsf{recv}\ y(w{:}\overline{A}); U(y, w)$.

■ (Case of $\mathcal{A} = \mathsf{sendty}\ x(T); R(x)$) By Lemma 10.6, there are E, F_x such that $P \equiv^{\mathsf{B}} E[F_x[\mathsf{sendty}\ x(T); R(x)]]\ \equiv^{\mathsf{B}}$ $E[\mathsf{cut}\ \{\mathsf{sendty}\ x(T); R\ |\overline{x}{:}\exists X.A\ [q]\ y{:}B]|\ S(y)\}]$ and $H = H'[x{:}\exists X.A\langle q, S(y)\rangle y{:}B]$. So (1) $enc(P) \Mapsto C$ by [S∃] where $C = (R(x), H[x{:}\{T/X\}A\langle q@\mathsf{ty}(T), S(y)\rangle y{:}B])^{wr}$. Then (2) $P \rightarrow^{\mathsf{B}} Q = E[\mathsf{cut}\ \{R(x)\ |\overline{x}{:}\{T/X\}A\ [q@\mathsf{ty}(T)]\ y{:}B]|\ S(y)\}]$ by [∃] and $Q \overset{\mathsf{cut}*}{\Mapsto} C$ for the two cases of $(\ldots)^{wr}$).

■ (Case of $\mathcal{A} = \mathsf{recvty}\ y(X); R(y)$) By Lemma 10.6, for some E, F_y we have $P \equiv^{\mathsf{B}} E[F_y[\mathsf{recvty}\ y(X); R(y)]]\ \equiv^{\mathsf{B}}$ $E[\mathsf{cut}\ \{S(x)\ |\overline{x}{:}C\ [q]\ y{:}\forall X.D]|\ \mathsf{recvty}\ y(X); R(y)\}]$ and $H = H'[x{:}C\langle q, S(x)\rangle y{:}\forall X.D]$ with $x \langle - \rangle y$ in read-mode. Since $\Delta \vdash q : \exists X.\overline{D} \triangleright C$, we have $q = \mathsf{ty}(T)@q'$ for some T, q', so $H = H'[x{:}C\langle \mathsf{ty}(T)@q', S(x)\rangle y{:}\forall X.D]$.

Then $enc(P) \Mapsto C$ (1) where $C = (\{T/X\}R(y), H[(x{:}A\langle q, S(x)\rangle y{:}\{T/X\}B)^{rw}])$ with C ready.

For (2), we have $P \rightarrow^{\mathsf{B}} Q$ with $Q = E[\mathsf{cut}\ \{S(x)\ |\overline{x}{:}A\ [q]\ y{:}\{T/X\}B]|\ \{T/X\}R(y)\}]$

or $Q = E[\mathsf{cut}\ \{S(x)\ |x{:}A\ [\mathsf{nil}]\ \overline{y}{:}\{T/X\}B]|\ \{T/X\}R(y)\}]$. As in $[\mathrm{S}\parr]$, in both cases we conclude $Q \overset{\mathsf{enc}*}{\Mapsto} C \overset{\mathsf{cut}*}{\Mapsto} enq(Q)$.

■ (Case of $\mathcal{A} = \mathsf{unfold}_\mu\ x; R(x)$) By Lemma 10.6, for contexts E, F_x we have $P \equiv^{\mathsf{B}} E[F_x[\mathsf{unfold}_\mu\ x; R(x)]]\ \equiv^{\mathsf{B}}$ $E[\mathsf{cut}\ \{\mathsf{unfold}_\mu\ x; R(x)\ |\overline{x}{:}\mu X.A\ [q]\ y{:}B]|\ S(y)\}]$ and $H = H'[x{:}\mu X.A\langle q, R(x)\rangle y{:}B]$, with $x \langle - \rangle y$ in write-mode.

Thus (1) $enc(P) \Mapsto C = (S(y), H'[x : \{\mu X.A/X\}A\langle q@\mathsf{step}, R(x)\rangle y : B])$ by [Sμ]. We then have (2) $P \rightarrow^{\mathsf{B}} Q = E[\mathsf{cut}\ \{R(x)\ |\overline{x} : \{\mu X.A/X\}A\ [q@\mathsf{step}]\ y : B]|\ S(y)\}]$ by [μ] and $Q \overset{\mathsf{enc}*}{\Mapsto} C \overset{\mathsf{cut}*}{\Mapsto} enq(Q)$.

■ (Case of $\mathcal{A} = \mathsf{rec}\ Y(u, \vec{w}); Q\ [y, \vec{z}]$) Abbreviate $R(y) = \mathsf{rec}\ Y(u, \vec{w}); Q\ [y, \vec{z}]$. By Lemma 10.6, there are E, F_x such that $P \equiv^{\mathsf{B}} E[R(y)]\ \equiv^{\mathsf{B}} E[F_x[\mathsf{cut}\ \{P(x)\ |\overline{x}{:}A\ [q]\ y{:}\nu X.B]|\ R(y)\}]]$ and $H = H'[x{:}A\langle q, P(x)\rangle y{:}\nu X.B]$ with $x \langle - \rangle y$ in read-mode. Since $\Delta \vdash q : \mu X.\overline{B} \triangleright A$, we must have $q = \mathsf{step}@q'$ and q step-terminated hence $q = \mathsf{step}$, do $H' = H[x{:}A\langle \mathsf{step}, P(x)\rangle y{:}\nu X.B]$. We consider the case $A+$. Then (1) $enc(P) \Mapsto (P(x), H'')^{p} = C$ where $H'' = H[x{:}A\langle \mathsf{nil}, U(y)\rangle y{:}\{\nu X.B/X\}B]$ and $U(y) = \{(\mathsf{rec}\ Y(u, \vec{w}); Q)/Y\}\{y, \vec{z}/u, \vec{w}\}Q$. We also have (2) $P \rightarrow^{\mathsf{B}} Q = E[\mathsf{cut}\ \{P(x)\ |\overline{x}{:}A\ [\mathsf{nil}]\ y{:}\{T/X\}B]|\ U(y)\}]$ and $Q \overset{\mathsf{enc}*}{\Mapsto} C \overset{\mathsf{cut}*}{\Mapsto} enq(Q)$. The case for $A-$ is similar. □

Theorem 5.6 (Soundness wrt CLLB). *Let $P \vdash^{\mathsf{B}} \emptyset$ and $enc(P)$ a ready SAM configuration. If $enc(P) \overset{*}{\Mapsto} C$ then there is Q such that $P \Rightarrow^{\mathsf{B}} Q$ and $Q \overset{\mathsf{enc}*}{\Mapsto} C$.*

Proof. By induction on the number n of SAM transition steps in $enc(P) \overset{*}{\Mapsto} C$.

Base case ($n = 0$): Trivial, $enc(P) \overset{0}{\Mapsto} enc(P)$ and $P = Q$. Inductive case ($n = 1 + n'$). Hence $enc(P)$ is live and by Lemma 5.5, there is $\mathcal{D}$ ready such that $enc(P) \Mapsto \mathcal{D} \overset{n'}{\Mapsto} C$, and P' such that $P \rightarrow^{\mathsf{B}} P'$ and $P' \overset{\mathsf{enc}*}{\Mapsto} \mathcal{D} \overset{\mathsf{cut}*}{\Mapsto} enc(P')$. By determinism of $\Mapsto$ we must have either (a) $\mathcal{D} \overset{n'\ \mathsf{enc}}{\Mapsto} C \overset{\mathsf{cut}*}{\Mapsto} enc(P')$ or (b) $\mathcal{D} \overset{m\ \mathsf{cut}}{\Mapsto} enc(P') \overset{m'}{\Mapsto} C$, with $n' = m + m'$. In (a) we conclude by letting $Q = P'$. For (b) by i.h., there is Q such that $P' \Rightarrow^{\mathsf{B}} Q$ and $Q \overset{\mathsf{enc}*}{\Mapsto} C$. Therefore, we conclude $P \Rightarrow^{\mathsf{B}} Q$ and $Q \overset{\mathsf{enc}*}{\Mapsto} C$. □

10.4 Proofs for Section 6: The Linear SAM for full CLL

We adapt prior definitions to deal with environments in configurations and closures, and formulate a revised version of the decomposition Lemma 10.6.

Lemma 10.7 (Encoding to Context - full SAM). *Let $P \vdash_{\mathsf{B}} \emptyset; \emptyset$ and $P \overset{\mathsf{enc}*}{\Mapsto} (\mathcal{E}, Q, H)$.*
Then $P \equiv_{\mathsf{B}} E[Q]$, and $E[\square] = F_1[F_2[\ldots[F_n[\square]]\ldots]]]$, for some E and
for every i, F_i is a one-hole cut context and $S_{a_i b_i} \in H$ a session record such that either:
(a) $F_i = \mathsf{cut}\ \{\square\ |\overline{x}_i{:}A\ [q]\ y_i{:}B|\ R\}$ and $\mathcal{E}(x_i) = a_i$ and $S_{a_i b_i} = \overline{a}_i{:}A\langle -, -, R\rangle b_i{:}B$ in write-mode, or
(b) $F_i = \mathsf{cut}\ \{R\ |\overline{x}_i{:}A\ [q]\ y_i{:}B|\ \square\}$ and $\mathcal{E}(y_i) = b_i$ and $S_{a_i b_i} = \overline{a}_i{:}A\langle -, -, R\rangle b_i{:}B$ in read-mode.
(c) $F_i = \mathsf{cut!}\ \{y.R\ |!x_i : A|\ \square\}]$ and $\mathcal{E}(x_i) = \mathsf{clos!}(y : \overline{A}, -, R)$.

Proof. By induction on $P \overset{\mathsf{ence}*}{\Mapsto} (\mathcal{E}, Q, H)$, we consider at each step one of the three cases of Definition 6.3. □

Lemma 6.5 (SAM-CLLB $\mathcal{E}$-Step Safety). *Let $P \vdash^{\mathsf{B}} \emptyset$ and $ence(P)$ a ready SAM configuration. If $ence(P)$ is live then (1) there is C ready such that $ence(P) \Mapsto C$ and (2) there is Q such that $P \rightarrow^{\mathsf{B}} Q$ and $Q \overset{\mathsf{ence}*}{\Mapsto} C \overset{\mathsf{cut}*}{\Mapsto} enc(Q)$.*

Proof. Assume $ence(P)$ live. We consider each case for $\mathcal{A}$, focusing here on the new cases for the exponentials. We illustrate the linear fragment with a detailed analysis of $\otimes$ and $\parr$.

■ (Case of [SCut!]) Not applicable, since [SCut!] is absorbed in $ence(P)$.

■ (Case of $\mathcal{A} = \mathsf{send}\ x(z.R(z)); U(x)$) By Lemma 10.7, for some E, F_x we have $P \equiv^{\mathsf{B}} E[F_x[\mathsf{send}\ x(z.R(z)); U(x)]]$ and $F_x[\square\ ] = \mathsf{cut}\ \{\square\ |\overline{x}{:}T \otimes A\ \ [q]\ \ y{:}B]|\ S(y)\}$. Thus $P \overset{\mathsf{ence}*}{\Mapsto} (\mathcal{G}, F_x[\mathsf{send}\ x(z.R(z)); U(x)], H') \overset{\mathsf{ence}}{\Mapsto} ence(P) = (\mathcal{E}, \mathcal{A}, H)$, where $\mathcal{E} \approx_{\mathcal{A}} \mathcal{G}\{a/x\}$, $\mathcal{F} \approx_{S(y)} \mathcal{G}\{b/y\}$, $qe \approx q^{\mathcal{G}}$ and $H = H'[a{:}T \otimes A\langle qe, \mathcal{F}, S(y)\rangle b{:}B]$ with $a\,\langle -\rangle\, b$ in write-mode.

Then, if $A+$ (a), $ence(P) \Mapsto C_1 = (\mathcal{E}, U(x), H'[a{:}A\langle qe@\mathsf{clos}(z, \mathcal{E}, R(z)), \mathcal{F}, S(y)\rangle b{:}B])$ by [S$\otimes$]; if $A-$ (b), $ence(P) \Mapsto C_2 = (\mathcal{F}, S(y), H'[a{:}A\langle qe@\mathsf{clos}(z, \mathcal{E}, R(z)), \mathcal{E}, U(x)\rangle b{:}B])$ by [S$\otimes$]. Notice that C_1 is ready ($a\,\langle -\rangle\, b$ in write-mode), C_2 is ready ($a\,\langle -\rangle\, b$ in read-mode). Hence we have (1). We now show (2).

For (a), $P \rightarrow^{\mathsf{B}} Q = E[\mathsf{cut}\ \{U(x)\ |\overline{x}{:}A+\ [q@\mathsf{clos}(z, R(z))]\ y{:}B]|\ S(y)\}]$ by [$\otimes$].

Let $F_x[\square\ ] = \mathsf{cut}\ \{\square\ |\overline{x}{:}A -\ \ [q]\ \ y{:}B]|\ S(y)\}$. We have $Q \overset{\mathsf{ence}*}{\Mapsto} (\mathcal{G}, F_x[U(x)], H') \overset{\mathsf{ence}}{\Mapsto} C_1$, since $\mathcal{E} \approx_{z.R(z)} \mathcal{G}$ and thus $q^{\mathcal{G}}@\mathsf{clos}(z, \mathcal{E}, R(z)) \approx (q@\mathsf{clos}(z, R(z)))^{\mathcal{G}}$. Then $C_1 \overset{\mathsf{cut}*}{\Mapsto} enc(Q)$ by Lemma 6.4(2). For (b), $P \rightarrow^{\mathsf{B}} Q = E[\mathsf{cut}\ \{U(x)\ |\overline{x}{:}A-\ [q@\mathsf{clos}(z, R(z))]\ y{:}B]|\ S(y)\}]$ by [$\otimes$]. Let $F'_y[\square\ ] = \mathsf{cut}\ \{U(x)\ |\overline{x}{:}A -\ \ [q]\ y{:}B]|\ \square\}$. We have $Q \overset{\mathsf{ence}*}{\Mapsto} (\mathcal{G}, F'_y[S(y)], H') \overset{\mathsf{ence}}{\Mapsto} C_2$, since $\mathcal{E} \approx_{z.R(z)} \mathcal{G}$ and thus $q^{\mathcal{G}}@\mathsf{clos}(z, \mathcal{E}, R(z)) \approx (q@\mathsf{clos}(z, R(z)))^{\mathcal{G}}$. Then $C_2 \overset{\mathsf{cut}*}{\Mapsto} ence(Q)$ by Lemma 6.4(2).

■ (Case of $\mathcal{A} = \mathsf{recv}\ y(w{:}\overline{A}); U(y, w)$) By Lemma 10.7, for contexts E, F_y we have $P \equiv^{\mathsf{B}} E[F_y[\mathsf{recv}\ y(w{:}\overline{A}); U(y, w)]]$ and $F_y[\square] = \mathsf{cut}\ \{S(x)\ |\overline{x}{:}C\ \ [q]\ \ y{:}\overline{A} \parr D]|\ \square\}$. Thus $P \overset{\mathsf{ence}*}{\Mapsto} (\mathcal{G}, F_y[\mathsf{recv}\ y(w{:}\overline{A}); U(y, w)], H') \overset{\mathsf{ence}}{\Mapsto} ence(P) = (\mathcal{F}, \mathcal{A}, H)$, $\mathcal{E} {\approx_{S(x)}} \mathcal{G}\{a/x\}$, $\mathcal{F} \approx_{\mathcal{A}} \mathcal{G}\{b/y\}$, $qe \approx q^{\mathcal{G}}$ and $H = H'[a{:}C\langle qe, \mathcal{E}, S(x)\rangle b{:}\overline{A} \parr D]$ with $a\,\langle -\rangle\, b$ in read-mode.

Since $\Gamma; \Delta \vdash q : A \otimes \overline{D} \triangleright C$, we have $q = \mathsf{clos}(z{:}A, R(z))@q'$. Hence $qe \approx \mathsf{clos}(z{:}A, \mathcal{G}', R(z))@qe'$, where $qe' \approx q'^{\mathcal{G}}$ and $\mathcal{G}' \approx_{z.R(z)} \mathcal{G}$. Wlog, assume $\mathcal{G}' = \mathcal{F}$, since $y \notin \mathsf{fn}(R(z))$. We prove (1,2) for the case (A+ and $q' \neq \mathsf{nil}$), other cases are like in the proof of Lemma 5.5. For (1) we have $ence(P) \Mapsto C$ by [S$\parr$] where $C = (\mathcal{F}\{c/z\}, R(z), H''')$, where $H'' = H'[a{:}C\langle qe', \mathcal{E}, S(x)\rangle b{:}D]$ and $H''' = H''[c{:}A\langle \mathsf{nil}, \mathcal{F}\{d/w\}, U(y, w)\rangle d{:}\overline{A}]$. Notice that C is ready ($a \langle - \rangle b$ in read-mode and $c \langle - \rangle d$ in write-mode).

For (2), let $P \to^{\mathsf{B}} Q = E[\ \mathsf{cut}\ \{S(x)\ |\overline{x}{:}C\ [q']\ y{:}D|\ C(y)\}]$ by [$\parr$], where $C(y) = \mathsf{cut}\ \{R(z)\ |\overline{z}{:}A\ [\mathsf{nil}]\ w{:}\overline{A}|\ U(y, w)\}$. Then $Q \overset{\mathsf{ence}*}{\Mapsto} (\mathcal{G}, \mathsf{cut}\ \{S(x)\ |\overline{x}{:}C\ [q']\ y{:}D|\ C(y)\}, H') \overset{\mathsf{ence}}{\Mapsto} (\mathcal{F}, C(y), H'') \overset{\mathsf{ence}}{\Mapsto} (\mathcal{F}\{c/z\}, R(z), H''') \overset{\mathsf{cut}*}{\Mapsto} ence(Q)$.

■ (Case of $\mathcal{A} = !x(z); R(z)$) By Lemma 10.7, $P \equiv^{\mathsf{B}} E[F_x[!x(z); R(z)]]$ and $F_x[\square\ ] = \mathsf{cut}\ \{\square\ |\overline{x}{:}!A\ [q]\ y{:}B]|\ S(y)\}$, for some contexts E, F_x. Thus $P \overset{\mathsf{ence}*}{\Mapsto} (\mathcal{G}, F_x[!x(z); R(z)], H') \overset{\mathsf{ence}}{\Mapsto} ence(P) = (\mathcal{E}, \mathcal{A}, H)$, where $\mathcal{E} \approx_{S(x)} \mathcal{G}\{a/x\}$, $\mathcal{F} \approx_{\mathcal{A}} \mathcal{G}\{b/y\}$, $qe \approx q^{\mathcal{G}}$, and $H = H'[a{:}!A\langle qe, \mathcal{F}, S(y)\rangle b{:}B]$ with $a \langle - \rangle b$ in write-mode.

By [S!], $ence(P) \Mapsto C = (\mathcal{F}, S(y), H'[a{:}\emptyset\langle qe@\mathsf{clos!}(z, \mathcal{E}, R(z)), \emptyset, 0\rangle b{:}B])$. C is ready ($a \langle - \rangle b$ in read-mode), so (1). For (2), let $F'_y[\square\ ] = \mathsf{cut}\ \{0\ |\overline{x}{:}\emptyset\ [q@\mathsf{clos!}(z, R(z))]\ y{:}B]|\ \square\}$ and $P \to^{\mathsf{B}} Q = E[F'_y[S(y)]]$ by [$\otimes$].

Then $Q \overset{\mathsf{ence}*}{\Mapsto} (\mathcal{G}, F'_y[S(y)], H') \overset{\mathsf{ence}}{\Mapsto} C$, since $\mathcal{G} \approx_{z.R(z)} \mathcal{E}$ and $qe@\mathsf{clos!}(z, \mathcal{E}, R(z)) \approx (q@\mathsf{clos}(z, R(z)))^{\mathcal{G}}$. Then $C \overset{\mathsf{cut}*}{\Mapsto} ence(Q)$ by Lemma 6.4(2).

■ (Case of $\mathcal{A} = ?y; Q$) By Lemma 10.7, for E, F_y we have $P \equiv^{\mathsf{B}} E[F_y[?y; Q]]$ and $F_y[\square] = \mathsf{cut}\ \{S(x)\ |\overline{x}{:}C\ [q]\ y{:}?B]|\ \square\}$. Thus $P \overset{\mathsf{ence}*}{\Mapsto} (\mathcal{G}, F_y[?y; Q], H') \overset{\mathsf{ence}}{\Mapsto} ence(P) = (\mathcal{F}, \mathcal{A}, H)$, where $H = H'[a{:}C\langle qe, \mathcal{E}, S(x)\rangle b{:}?B]$ with $a \langle - \rangle b$ in read-mode, $\mathcal{E} \approx_{S(x)} \mathcal{G}\{a/x\}$, $\mathcal{F} \approx_{\mathcal{A}} \mathcal{G}\{b/y\}$ and $qe \approx q^{\mathcal{G}}$. Since $\Gamma; \vdash q : !\overline{B} \triangleright A$, we have $q = \mathsf{clos!}(z, R(z))$ and $A = \emptyset$ and $S(x) = 0$, and $H = H'[x{:}\emptyset\langle \mathsf{clos!}(z, \mathcal{G}', R(z)), 0\rangle y{:}?B]$ where $\mathcal{G}' \approx_{z.R(z)} \mathcal{G}$. Wlog., assume $\mathcal{G}' = \mathcal{F}$, since $y \notin \mathsf{fn}(R(z))$. For (1), $ence(P) \Mapsto C = (\mathcal{F}\{\mathsf{clos!}(z, \mathcal{F}, R(z))/y\}, Q, H')$, with C ready. For (2), let $P \to^{\mathsf{B}} Q \equiv E[\mathsf{cut!}\ \{z.R\ |!y|\ Q\}]$ by [?]. We have $Q \overset{\mathsf{ence}*}{\Mapsto} (\mathcal{G}, \mathsf{cut!}\ \{z.R\ |!y|\ Q\}, H') \overset{\mathsf{ence}}{\Mapsto} C$, since $\mathcal{F} \approx_{z.R(z)} \mathcal{G}$, and $\mathcal{F}\{\mathsf{clos!}(z, \mathcal{F}, R(z))/y\} = \mathcal{G}\{\mathsf{clos!}(z, \mathcal{F}, R(z))/y\}$. We conclude $C \overset{\mathsf{cut}*}{\Mapsto} ence(Q)$ by Lemma 6.4(2).

■ (Case of $\mathcal{A} = \mathsf{call}\ y(w); U(w)$) By Lemma 10.7, for contexts E, F_y, E' we have $P \equiv^{\mathsf{B}} E[F_y[E'[\mathsf{call}\ y(w); U(w)]]]$ and $F_y[\square] = \mathsf{cut!}\ \{z.R\ |!y : A|\ \square\}$ and $P \overset{\mathsf{ence}*}{\Mapsto} ence(P) = (\mathcal{G}, \mathcal{A}, H)$, where $\mathcal{G}(y) = \mathsf{clos!}(z, \mathcal{G}', R(z))$ for some $\mathcal{G}' \approx_{z.R(z)} \mathcal{G}$. We have $ence(P) \Mapsto C = (\mathcal{E}, U(w), H[a{:}A\langle \mathsf{nil}, \mathcal{G}'\{b/z\}, R(z)\rangle b{:}\overline{A}])^{\mathsf{P}}$ by [Scall], where $\mathcal{E} = \mathcal{G}\{a/w\}$.

We branch on A polarity. If A+, then $C = (\mathcal{E}, U(w), H[a{:}A\langle \mathsf{nil}, \mathcal{G}'\{b/z\}, R(z)\rangle b{:}\overline{A}])$. C is ready, with $a \langle - \rangle b$ in write-mode, hence (1). Let $P \to^{\mathsf{B}} Q \equiv E[F_y[E'[\mathsf{cut}\ \{U(w)\ |\overline{w}{:}A[\mathsf{nil}]z{:}\overline{A}|\ R(z)\}]]]$ by [call].

Then $Q \overset{\mathsf{ence}*}{\Mapsto} (\mathcal{G}, \mathsf{cut}\ \{U(w)\ |\overline{w}{:}A[\mathsf{nil}]z{:}\overline{A}|\ R\}, H') \overset{\mathsf{ence}}{\Mapsto} C$, since $\mathcal{G}'\{b/z\} \approx_{R(z)} \mathcal{G}\{b/z\}$. For (2), $C \overset{\mathsf{cut}*}{\Mapsto} ence(Q)$ by Lemma 6.4(2).

If A−, then $ence(P) \Mapsto C = (\mathcal{G}'\{b/z\}, R(z), H[b{:}\overline{A}\langle \mathsf{nil}, \mathcal{E}, U(w)\rangle a{:}A])$. Then (1) C is ready, with $b \langle - \rangle a$ in write-mode. Let $P \to^{\mathsf{B}} Q \equiv E[F_y[E'[\mathsf{cut}\ \{R(z)\ |\overline{z}{:}\overline{A}\ [\mathsf{nil}]\ w{:}A|\ U(w)\}]]]$ by [call].

Then $Q \overset{\mathsf{ence}*}{\Mapsto} (\mathcal{G}, \mathsf{cut}\ \{U(w)\ |\overline{w}{:}A\ [\mathsf{nil}]\ z{:}\overline{A}|\ R(z)\}, H') \overset{\mathsf{ence}}{\Mapsto} C$, since $\mathcal{G}'\{b/z\} \approx_{R(z)} \mathcal{G}\{b/z\}$. For (2), $C \overset{\mathsf{cut}*}{\Mapsto} ence(Q)$ by Lemma 6.4(2). □

10.5 Proofs of Section 7: Concurrent Semantics

We present the proof of the main soundness and progress Lemma 7.4 for the Concurrent Linear SAM, from which Theorems 7.5 and 7.6 easily follow. First we define the encoding of CLLB processes with annotated concurrent cuts. Such annotation is silent for any purpose other than the concurrent execution strategy in the CSAM. For simplicity, we extend the basic Linear SAM of Section 5 (no exponentials), but with concurrent mix and cut constructs.

$$(\{\mathbf{0}\} \uplus \mathcal{M}, H) \overset{\mathsf{encc}}{\Mapsto} (\mathcal{M}, H) \qquad\qquad [\mathrm{C}\mathbf{0}]$$

$$(\{P\} \uplus \mathcal{M}, H') \overset{\mathsf{encc}}{\Mapsto} (\{Q\} \uplus \mathcal{M}, H') \qquad \text{if } enc(P, H) \overset{\mathsf{enc}}{\Mapsto} (Q, H') \qquad [\mathrm{CThr}]$$

$$(\mathsf{cpar}\ \{P \mid\mid Q\} \ \uplus \mathcal{M}, H) \overset{\mathsf{encc}}{\Mapsto} (\{P, Q\} \uplus \mathcal{M}, H) \qquad\qquad [\mathrm{CMix}]$$

$$(\mathsf{ccut}\ \{P \,|\overline{x} : A[q]y : B|\, Q\} \uplus \mathcal{M}, H) \overset{\mathsf{encc}}{\Mapsto} (\{P, Q\} \uplus \mathcal{M}, H[x\,\langle q\rangle\, y]) \qquad\qquad [\mathrm{CCut}]$$

Lemma 10.8 (Encoding to Context - concurrent SAM). *Let $P \vdash_{\mathsf{B}} \emptyset$ and $P \overset{\mathsf{encc}*}{\Mapsto} (\tilde{A}, H)$. Then, for every $A_i \in \mathcal{A}$ there is $E_A[\square] = F_1[F_2[\ldots[F_n[\square]]\ldots]]]$ such that $P \equiv E[A_i]$ and for every i, F_i is a one-hole cut context and $S_{x_i y_i} \in H$ a session record such that either:*
(a) $F_i \equiv \mathsf{cut}\ \{\square\ |x_i{:}A\ [q]\ y_i{:}B|\ R\}$ and $S_{x_i y_i} = x_i{:}A\langle q, R\rangle y_i{:}B$ in write-mode, or
(b) $F_i \equiv \mathsf{cut}\ \{R\ |\overline{x}_i{:}A\ [q]\ y_i{:}B|\ \square\}$ and $S_{x_i y_i} = \overline{x}_i{:}A\langle q, R\rangle y_i{:}B$ in read-mode.
(c) $Fi \equiv \mathsf{ccut}\ \{\square\ |\overline{x}_i{:}A\ [q]\ y_i{:}B|\ R\}$ and $S_{x_i y_i} = x_i{:}A\,\langle q\rangle\, y_i{:}B$.
(d) $Fi \equiv \mathsf{ccut}\ \{R\ |\overline{x}_i{:}A\ [q]\ y_i{:}B|\ \square\}$ and $S_{x_i y_i} = x_i{:}A\,\langle q\rangle\, y_i{:}B$.
(e) $F_i \equiv \mathsf{cpar}\ \{\square \mid\mid R\}$.

Proof. By induction on $P \overset{\mathsf{encc}*}{\Mapsto} (Q, H)$, we consider at each step one of the cases in Definition 7.2. □

The proof follows the structure leading to Theorem 4.10, where the liveness Lemma is now based on a notion of observation (cf. Figure 12), adapted to the present setting of concurrent actions on concurrent session records. Intuitively, $P \downarrow_x^H$ holds if process P is about to execute an action on a concurrent session endpoint.

Definition 10.9. We denote by $P \downarrow_x^H$ the assertion that $P \downarrow_x$ where either $x\,\langle q\rangle\, y \in H$ or $y\,\langle q\rangle\, x \in H$.

Lemma 10.10 (Liveness-s). *Let $P \vdash^{\mathsf{B}} \emptyset$ and $P \overset{\mathsf{encc}*}{\Mapsto} \mathcal{D} = (\tilde{\mathcal{A}}, H)$ where $\mathcal{D}$ ready. Then either*

1. *There is C ready such that $\mathcal{D} \Mapsto C$ and Q such that $P \rightarrow^{\mathsf{B}} Q$ and $Q \overset{\mathsf{encc}*}{\Mapsto} C \overset{\mathsf{cut}*}{\Mapsto} encc(Q)$, or*
2. *For all $A_i \in \tilde{\mathcal{A}}$, we have $A_i \downarrow_x^{H'}$.*

Proof. If all $A_i \in \tilde{\mathcal{A}}$ are endpoints of concurrent session records in H, we conclude (2). Otherwise, $\tilde{\mathcal{A}} = A(x) \uplus \tilde{\mathcal{A}}'$ for some action $A(x)$ where the subject x the endpoint of a sequential session record. By Lemma 10.8 there are contexts E, F_x such that $P \equiv^{\mathsf{B}} E[F_x[A(x)]]$. Then, as for Lemma 5.5 for $(\mathcal{A}, H) \Mapsto C$, we prove that $(A(x) \uplus \tilde{\mathcal{A}}', H) \Mapsto C = (R \uplus \tilde{\mathcal{A}}', H')$, and $F_x[A(x)] \rightarrow_{\mathsf{B}} R$ for some R, so $P \rightarrow_{\mathsf{B}} E[R] = Q$ and $encc(Q) \overset{\mathsf{encc}*}{\Mapsto} C \overset{\mathsf{cut}*}{\Mapsto} encc(Q)$, concluding (1). □

Lemma 10.11 (Liveness). *Let $P \vdash \emptyset$ and $P \overset{\mathsf{encc}*}{\Mapsto} (\mathcal{N}, H) \overset{\mathsf{encc}*}{\Mapsto} \mathcal{D} = (\tilde{\mathcal{A}}, H')$ where $\mathcal{D}$ ready. Then either*

1. *There is C ready such that $\mathcal{D} \Mapsto C$ and Q such that $P \rightarrow^{\mathsf{B}} Q$ and $Q \overset{\mathsf{encc}*}{\Mapsto} C \overset{\mathsf{cut}*}{\Mapsto} encc(Q)$, or*
2. *For all $P \in \mathcal{N}$, there is $A_i \in \tilde{\mathcal{A}}^P$, such that $A_i \downarrow_x^{H'}$ with $x \in \mathsf{fn}(P)$.*

Proof. By induction on $(\mathcal{N}, H) \overset{\mathsf{encc}*}{\Mapsto} \mathcal{D}$. By Lemma 7.3 (1), both $(\mathcal{N}, H)$ and $\mathcal{D}$ are ready.

■ (Base Case) We have $(\mathcal{N}, H) = (\tilde{\mathcal{A}}, H)$. The conclusion follows from Lemma 10.10, since $\mathcal{N} = \tilde{\mathcal{A}}$.

■ (Case of [C$\mathbf{0}$]) By the i.h..

■ (Case of [CThr]) We have $C = (\mathcal{N}, H) \overset{\mathsf{encc}}{\Mapsto} (\{Q\} \uplus \mathcal{M}, H') \overset{\mathsf{encc}*}{\Mapsto} (\tilde{\mathcal{A}}, H'') = C'$, where $\mathcal{N} = \{R\} \uplus \mathcal{M}$ and $(R, H) \overset{\mathsf{enc}}{\Mapsto} (Q, H')$ where $R = F[Q]$ for a basic (non-concurrent) cut context F. By i.h., we have either ((1)) $\mathcal{D} \Mapsto C$ or ((2)) for all for all $S \in \{Q\} \uplus \mathcal{M}$, there is $A \in \mathcal{A}^S$, $A \downarrow_z^{H'}$ with $z \in \mathsf{fn}(S)$. In case ((1)) we conclude (1). In case ((2)) since $F[]$ is a non-concurrent cut, it does not bind z, hence if $A \in \mathcal{A}^Q$, $A \downarrow_z^{H'}$ with $z \in \mathsf{fn}(Q)$ then $A \in \mathcal{A}^R$, $A \downarrow_z^{H'}$ with $z \in \mathsf{fn}(R)$. Hence we conclude (2).

■ (Case of [CMix]) We have $C = (\mathcal{N}, H) \overset{\text{encc}}{\Mapsto} (\{U_1, U_2\} \uplus \mathcal{M}, H) \overset{\text{encc}*}{\Mapsto} (\tilde{\mathcal{A}}, H') = C'$, where $\mathcal{N} = U \uplus \mathcal{M}$, with $U = \mathsf{cpar}\ \{U_1 \mid\mid U_2\}$. By i.h., either ((1)) $C' \Mapsto C$ or ((2)) for all $S \in \{U_1, U_2\} \uplus \mathcal{M}$, there is $A \in \tilde{\mathcal{A}}^S$, where $A \downarrow_z^{H'}$ for $z \in \mathsf{fn}(S)$. For ((1)) we conclude (1). In case ((2)), (2) also immediately follows from the i.h., since, e.g., from $A \in \mathcal{A}_1^U$, $A \downarrow_z^{H'}$ with $z \in \mathsf{fn}(U_1)$ we conclude $A \in \mathcal{A}^R$, $A \downarrow_z^{H'}$ with $z \in \mathsf{fn}(U)$.

■ (Case of [CCut]) We have $C = (\mathcal{N}, H) \overset{\text{encc}}{\Mapsto} (\{U_1, U_2\} \uplus \mathcal{M}, H[x\,\langle q\rangle\,y]) \overset{\text{encc}*}{\Mapsto} (\tilde{\mathcal{A}}, H') = C'$, where $\mathcal{N} = U \uplus \mathcal{M}$, with $U = \mathsf{ccut}\ \{U_1 \mid \overline{x} : A[q]y : B \mid U_2\}$. By i.h., either ((1)) $C' \Mapsto C$ or ((2)) for all $S \in \{U_1, U_2\} \uplus \mathcal{M}$, there is $A \in \tilde{\mathcal{A}}^S$, where $A \downarrow_z^{H'}$ for $z \in \mathsf{fn}(S)$. For ((1)) we conclude (1).

Otherwise, we assume ((2)) and consider the cases $A+$ and not $A+$.

(Case A positive) Then $x \in \mathsf{fn}(U_1)$, so $U_1 \neq 0$. Then (i.h.) there is $A \in \tilde{\mathcal{A}}^{U_1}$ with $A \downarrow_z^{H'}$. If $z \neq x$ then $A \downarrow_z^{H'}$ with $A \in \tilde{\mathcal{A}}^U$, hence ((2)). If $z = x$, then $A(x)$ must be a positive action, so we have $(A, H') \Mapsto (P', H'') = C$, where $x\,\langle q\rangle\,y$ in H' mutates to $x\,\langle q@v\rangle\,y \in H''$ in H'', by one of the (forwarding or positive) transition rules for the concurrent SAM (Figure 22). By Lemma 10.8 there are contexts E, F_x such that $P \equiv^{\mathsf{B}} E[F_x[A(x)]]$. As in Lemma 5.5 there is R such that $F_x[A(x)] \to_{\mathsf{B}} R$, so $P \to_{\mathsf{B}} E[R] = Q$ where $\mathit{encc}(Q) \overset{\text{encc}*}{\Mapsto} C \overset{\text{cut}*}{\Mapsto} \mathit{encc}(Q)$. We thus conclude ((1)).

(Case A not positive) Then type A is either negative or $A = \emptyset$. Since B is negative, we must have $y \in \mathsf{fn}(U_2)$ and thus (i.h.) there is $B \in \tilde{\mathcal{A}}^{U_2}$ with $B \downarrow_w^{H'}$. If $w \neq y$ then $B \downarrow_w^{H'}$ with $B \in \tilde{\mathcal{A}}^U$, hence ((2)). If $w = y$, then $B(y)$ is a negative action. Since $\Delta \vdash q : \overline{B} \triangleright A$ with A negative or $\emptyset$, we must have $q \neq \mathsf{nil}$, with $q = v@q'$ and v a value of the appropriate type for the action $B(y)$. Hence $(B, H') \Mapsto (P', H'') = C$ by one of the (forwarding or negative) transition rules for the concurrent SAM (Figure 22) and we conclude ((1)) as above. □

Lemma 7.4 (CSAM-CLLB Step Safety). *Let $P \vdash^{\mathsf{B}} \emptyset$ and $\mathit{encc}(P)$ a ready SAM configuration. If $\mathit{encc}(P)$ is live then (1) there is C ready such that $\mathit{ence}(P) \Mapsto C$ and (2) there is Q such that $P \to^{\mathsf{B}} Q$ and $Q \overset{\text{encc}*}{\Mapsto} C \overset{\text{cut}*}{\Mapsto} \mathit{ence}(Q)$.*

Proof. By Lemma 10.11, by setting $(\{P\}, \emptyset) = (\mathcal{N}, H)$, and noting that Lemma 10.11(2) cannot hold for $\mathsf{fn}(P) = \emptyset$. □