On the Fair Termination of Client-Server Sessions

Luca Padovani
University of Camerino, Italy

Abstract

Client-server sessions are based on a variation of the traditional interpretation of linear logic propositions as session types in which non-linear channels (those regulating the interaction between a pool of clients and a single server) are typed by *coexponentials* instead of the usual exponentials. Coexponentials enable the modeling of racing interactions, whereby clients compete to interact with a single server whose internal state (and thus the offered service) may change as the server processes requests sequentially. In this work we present a fair termination result for CSLL^∞, a core calculus of client-server sessions. We design a type system such that every well-typed term corresponds to a valid derivation in $\mu\mathsf{MALL}^\infty$, the infinitary proof theory of linear logic with least and greatest fixed points. We then establish a correspondence between reductions in the calculus and principal reductions in $\mu\mathsf{MALL}^\infty$. Fair termination in CSLL^∞ follows from cut elimination in $\mu\mathsf{MALL}^\infty$.

2012 ACM Subject Classification Theory of computation → Linear logic; Theory of computation → Process calculi; Theory of computation → Program analysis

Keywords and phrases client-server sessions, linear logic, fixed points, fair termination, cut elimination

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.5

Acknowledgements The author is grateful to the anonymous reviewers of the TYPES'22 post-proceedings for their helpful comments, observations and suggestions of related work.

1 Introduction

Session types [14, 15, 16] are descriptions of communication protocols enabling the static enforcement of a variety of safety and liveness properties, including the fact that communication channels are used according to their protocol (*fidelity*), that processes do not get stuck (*deadlock freedom*), that pending communications are eventually completed (*livelock freedom*), that sessions eventually end (*termination*). It is possible to trace a close correspondence between session types and propositions of linear logic, and between the typing rules of a session type system and the proof rules of linear logic [21, 6, 19]. This correspondence provides session type theories with a solid logical foundation and enables the application of known results concerning linear logic proofs into the domain of communicating protocols. One notable example is *cut elimination*: the fact that every linear logic proof can be reduced to a form that does not make use of the cut rule means that the process described by the proof can be reduced to a form in which no pending communication is present, provided that there is a good correspondence between cut reductions in proofs and reductions in processes.

The development of session type systems based on linear logic also poses some challenges with respect to their ability to cope with "real-world" scenarios. An example, which is the focus of this work, is the modeling of the interactions between a *pool of clients* and a *single server*. By definition, a server is a process that can handle an unbounded number of requests made by clients. In a session type system based on linear logic, it is natural to associate the channel from which a server accepts client requests with a type of the form $!T$, indicating the unlimited availability of a service with type T. In fact, it was observed early on [11] that the meaning of the "of course" modality $!T$ could be informally expressed by the equation

$$!T \cong \mathbf{1} \mathbin{\&} T \mathbin{\&} (!T \otimes !T)$$

28th International Conference on Types for Proofs and Programs (TYPES 2022).
Editors: Delia Kesner and Pierre-Marie Pédrot; Article No. 5; pp. 5:1–5:21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

which could be read "as many copies of T as the clients require". While appealing from a theoretical point of view, the association between the concept of server and the "of course" modality is both unrealistic and imprecise. First of all, it models the "unlimited" availability of the server by means of unlimited *parallel* copies of the server, each copy dealing with a single request, rather than by a *single* process that is capable of handling an unlimited number of requests *sequentially*. Second, it fails to capture the fact that each connection between a client and the server may alter the server's internal state, in such a way that different connections may potentially affect each other.

These considerations have led Qian et al. [20] to develop CSLL (for "Client-Server Linear Logic"), a session type system based on linear logic which includes the *coexponential* modalities $¿T$ and $¡T$ whose meaning can be (informally) expressed by the equations

$$¿T \cong \mathbf{1} \oplus T \oplus (¿T \otimes ¿T) \qquad \text{and} \qquad ¡T \cong \bot \mathbin{\&} T \mathbin{\&} (¡T \parr ¡T) \tag{1}$$

according to which a server that behaves as $¡T$ offers T as many times as necessary to satisfy all client requests, but it does so sequentially and in some (unspecified) order. Qian et al. [20] show that well-typed CSLL processes are deadlock free, but they leave a proof of termination to future work conjecturing that it could be quite involved. A proof of this property is valuable since termination (combined with deadlock freedom) implies livelock freedom.

In this paper we attack the problem of establishing a termination result for CSLL. Instead of providing an *ad hoc* proof, we attempt to reduce the termination problem for CSLL to the cut elimination property of a known logical system. To this aim, we propose a variation of CSLL called CSLL^∞ that is in close relationship with $\mu\mathsf{MALL}^\infty$ [3, 8, 2], the infinitary proof theory of multiplicative-additive linear logic with least and greatest fixed points. The basic idea is to encode the coexponentials in CSLL^∞'s type system as fixed points in $\mu\mathsf{MALL}^\infty$ following their expected meaning (Equation (1)). At this point, the cut elimination property of $\mu\mathsf{MALL}^\infty$ should allow us to deduce that well-typed CSLL^∞ processes do not admit infinite reduction sequences. As it turns out, we are unable to follow this plan of action in full. The problem is that some reductions in CSLL^∞ do not correspond to cut reduction steps in $\mu\mathsf{MALL}^\infty$. More specifically, even though clients are queued into client pools, they should be able to reduce in any order, independently of their position in the queue. This independent reduction of the clients in the same pool is not matched by the sequence of cut reduction steps that are performed in the cut elimination proof of $\mu\mathsf{MALL}^\infty$. Still, the cut elimination property of $\mu\mathsf{MALL}^\infty$ allows us to prove a useful result, namely that every well-typed CSLL^∞ process is *fairly terminating*. Fair termination [13, 10] is weaker than termination since it does not rule out the existence of infinite reduction sequences. However, it guarantees that every *fair* and maximal reduction sequence of a well-typed CSLL^∞ process is finite, under a suitable fairness assumption. In particular, fair termination is strong enough (when combined with deadlock freedom) to guarantee livelock freedom.

The adoption of $\mu\mathsf{MALL}^\infty$ as logical foundation for CSLL^∞ has another advantage. In the original presentation of CSLL [20] the process calculus is equipped with an unconventional operational semantics whereby reductions can occur underneath prefixes and prefixes may be moved around crossing restrictions, parallel compositions and other (unrelated) prefixes. This semantics is justified to keep the process reduction rules and the cut reduction rules sufficiently aligned, so that the cut elimination property in the logic can be reflected to some valuable property in the calculus, such as deadlock freedom. In contrast, CSLL^∞ features an entirely conventional reduction semantics. We can afford to do so because $\mu\mathsf{MALL}^\infty$ is an *infinitary* proof system in which the cut elimination property is proved bottom-up by *reducing outermost cuts first*. This reduction strategy matches the ordinary reduction

Table 1 Syntax of CSLL$^\infty$.

$P,Q ::= \mathsf{A}\langle\overline{x}\rangle$	invocation	$\mid (x)(P \mid Q)$	parallel composition		
$\mid\ \ \mathsf{fail}\,x$	failure	$\mid$ ¿$x[]$	empty pool		
$\mid\ \ \mathsf{wait}\,x.P$	wait	$\mid \mathsf{close}\,x$	close		
$\mid\ \ x(y).P$	input	$\mid x[y](P \mid Q)$	output		
$\mid\ \ \mathsf{case}\,x\{P,Q\}$	branch	$\mid \mathsf{in}_i\,x.P$	select	$i \in \{1,2\}$	
$\mid$ ¡$x(y)\{P,Q\}$	server	$\mid$ ¿$x[y].P :: Q$	client pool		

semantics of any process calculus in which reductions happen at the outermost levels of processes. In the end, since the reduction semantics of CSLL$^\infty$ is stricter than that of CSLL, the deadlock freedom and the fair termination results we prove for CSLL$^\infty$ are somewhat stronger than their counterparts in the context of CSLL.

Structure of the paper. Section 2 describes syntax and semantics of CSLL$^\infty$ and defines the notion of fairly terminating process. We develop the type system for CSLL$^\infty$ in Section 3. In Section 4 we recall the key elements of μMALL$^\infty$, before addressing the proof that well-typed CSLL$^\infty$ processes fairly terminate in Section 5. Section 6 revisits an example of non-deterministic server given by Qian et al. [20] in our setting. We summarize our results and further compare CSLL$^\infty$ with CSLL [20] and other related work in Section 7. Some proofs and definitions have been moved into Appendix A.

2 Syntax and Semantics of CSLL$^\infty$

In this section we define syntax and semantics of CSLL$^\infty$, a calculus of sessions in which servers handle client requests sequentially. The syntax of CSLL$^\infty$ makes use of an infinite set $\mathcal{V}$ of *channels* ranged over by x, y and z and a set $\mathcal{P}$ of *process names* ranged over by A, B, and so on. In CSLL$^\infty$ channels are of two kinds (which will be distinguished by their type): *session channels* connect two communicating processes; *shared channels* connect an unbounded number of clients with a single server. The structure of terms is given by the grammar in Table 1 and their meaning is informally described below. The term $(x)(P \mid Q)$ represents the parallel composition of P and Q connected by the restricted channel x, which can be either a session channel or a shared channel. The term $\mathsf{fail}\,x$ represents a process that signals a failure on channel x. The term $\mathsf{close}\,x$ models the closing of a session, whereas $\mathsf{wait}\,x.P$ models a process that waits for x to be closed and then continues as P. The term $x[y](P \mid Q)$ models a process that creates a new channel y, sends y over x, uses y as specified by P and x as specified by Q. The term $x(y).P$ models a process that receives a channel y from x and then behaves as P. The term $\mathsf{in}_i\,x.P$ models a process that sends the label in_i over x and then behaves as P. In this work we only consider two labels in_1 and in_2, although it is common to allow for an arbitrary set of atomic labels. Dually, the term $\mathsf{case}\,x\{P_1,P_2\}$ models a process that waits for a label in_i from x and then behaves according to P_i. The term ¿$x[]$ models the empty pool of clients connecting with a server on the shared channel x, whereas the term ¿$x[y].P :: Q$ models a client pool consisting of a client that connects with a server on channel x and behaves as P and another client pool Q. Occasionally we write ¿$x[y].P$ instead of ¿$x[y].P ::$ ¿$x[]$. The term ¡$x(y)\{P,Q\}$ models a server that waits for connections on the shared channel x. If a new connection y is established, the server continues as P. If no clients are left connecting on x, the service on x is terminated and the process continues as Q. Finally, a term $\mathsf{A}\langle\overline{x}\rangle$ represents the invocation of the process named

Table 2 Structrual pre-congruence and reduction semantics of CSLL^∞.

[S-PAR-COMM]	$(x)(P \mid Q) \preccurlyeq (x)(Q \mid P)$	
[S-POOL-COMM]	$¿x[y].P :: ¿u[v].Q :: R \preccurlyeq ¿u[v].Q :: ¿x[y].P :: R$	
[S-PAR-ASSOC]	$(x)(P \mid (y)(Q \mid R)) \preccurlyeq (y)((x)(P \mid Q) \mid R)$	$x \in \mathsf{fn}(Q) \setminus \mathsf{fn}(R), y \notin \mathsf{fn}(P)$
[S-POOL-PAR]	$¿x[y].P :: (z)(Q \mid R) \preccurlyeq (z)(¿x[y].P :: Q \mid R)$	$x \in \mathsf{fn}(Q), z \notin \mathsf{fn}(¿x[y].P)$
[S-PAR-POOL]	$(z)(¿x[y].P :: Q \mid R) \preccurlyeq ¿x[y].P :: (z)(Q \mid R)$	$z \notin \mathsf{fn}(¿x[y].P)$
[S-CALL]	$\mathsf{A}\langle \overline{x} \rangle \preccurlyeq P$	$\mathsf{A}(\overline{x}) \triangleq P$
[R-CLOSE]	$(x)(\mathsf{close}\, x \mid \mathsf{wait}\, x.P) \to P$	
[R-COMM]	$(x)(x[y](P \mid Q) \mid x(y).R) \to (y)(P \mid (x)(Q \mid R))$	
[R-CASE]	$(x)(\mathsf{in}_i\, x.P \mid \mathsf{case}\, x\{Q_1, Q_2\}) \to (x)(P \mid Q_i)$	
[R-DONE]	$(x)(¿x[] \mid ¡x(y)\{P, Q\}) \to Q$	
[R-CONNECT]	$(x)(¿x[y].P :: Q \mid ¡x(y)\{R_1, R_2\}) \to (y)(P \mid (x)(Q \mid R_1))$	
[R-PAR]	$(x)(P \mid R) \to (x)(Q \mid R)$	$P \to Q$
[R-POOL]	$¿x[y].R :: P \to ¿x[y].R :: Q$	$P \to Q$
[R-STRUCT]	$P \to Q$	$P \preccurlyeq P' \to Q' \preccurlyeq Q$

A with arguments $\overline{x}$. We assume that each process name is associated with a unique global definition of the form $\mathsf{A}(\overline{x}) \triangleq P$. The notation $\overline{e}$ is used throughout the paper to represent possibly empty sequences $e_1, \dots, e_n$ of various entities.

The notions of free and bound names are defined in the expected way. Note that the output operations $x[y](P \mid Q)$ and $¿x[y].P :: Q$ bind y in P but not in Q. We write $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$ for the sets of free and bound names in P, we identify processes up to renaming of bound channel names and we require $\mathsf{fn}(P) = \{\overline{x}\}$ for each global definition $\mathsf{A}(\overline{x}) \triangleq P$.

The operational semantics of CSLL^∞ is given by a structural precongruence relation $\preccurlyeq$ and a reduction relation $\to$, both defined in Table 2 and described below. Rules [S-PAR-COMM] and [S-POOL-COMM] state the expected commutativity of parallel and pool compositions. In particular, [S-POOL-COMM] allows clients in the same queue to swap positions, modeling the fact that the order in which they connect to the server is not deterministic. Rule [S-PAR-ASSOC] models the associativity of parallel composition. The side conditions make sure that no channel is captured ($y \notin \mathsf{fn}(P)$) or left dangling ($x \notin \mathsf{fn}(R)$) and that parallel processes remain connected ($x \in \mathsf{fn}(Q)$). The rules [S-POOL-PAR] and [S-PAR-POOL] deal with the mixed associativity between parallel and pool compositions. The side conditions ensure that no bound name leaves its scope and that parallel processes remain connected. Finally, [S-CALL] unfolds a process invocation to its definition.

Concerning the reduction relation, rule [R-CLOSE] models the closing of a session, rule [R-COMM] models the exchange of a channel and [R-CASE] that of a label. Rule [R-CONNECT] models the connection of a client with a server, whereas [R-DONE] deals with the case in which there are no clients left. Finally, [R-PAR] and [R-POOL] close reductions under parallel compositions and client pools whereas [R-STRUCT] allows reductions up to structural pre-congruence.

Hereafter we write $\Rightarrow$ for the reflexive, transitive closure of $\to$, we write $P \to$ if $P \to Q$ for some Q and $P \not\to$ if not $P \to$. Later on we will also use a restriction of CSLL^∞ dubbed $\mathsf{CSLL}^\infty_{\mathsf{det}}$ whose reduction relation, denoted by $\to_{\mathsf{det}}$, is obtained by removing the rules [S-POOL-COMM], [S-POOL-PAR], [S-PAR-POOL] and [R-POOL] (all those with "POOL" in their name) from $\to$. In essence, $\mathsf{CSLL}^\infty_{\mathsf{det}}$ is a more deterministic version of CSLL^∞ in which clients are forced to connect and reduce in the order in which they appear in client pools. Also, clients are no longer allowed to cross restricted channels.

▶ **Example 1.** We illustrate the features of CSLL^∞ by modeling a pool of clients that compete to access a shared resource, represented as a simple *lock*. When one client manages to acquire the lock, meaning that it has gained access to the resource, it prevents other clients from accessing the resource until the resource is released. We model the lock with this definition:

$$\mathsf{Lock}(x, z) \triangleq\ ¡x(y)\{\mathsf{wait}\, y.\mathsf{Lock}\langle x, z\rangle, \mathsf{close}\, z\}$$

The lock is a server waiting for connections on the shared channel x, whereas each user is a client of the lock connecting on x. When a connection is established, the server waits until the resource is released, which is signalled by the termination of the session y, and then makes itself available again to handle further requests.

The following process models the concurrent access to the lock by two clients:

$$(x)(¿x[u].\mathsf{close}\, u :: ¿x[v].\mathsf{close}\, v :: ¿x[] \mid \mathsf{Lock}\langle x, z\rangle)$$

The order in which requests are handled by Lock is non-deterministic because of [s-pool-comm]. In this oversimplified example the users are indistinguishable and so non-determinism does not prevent the system to be confluent. In Section 6 we will see a more interesting example in which confluence is lost. This kind of interaction is typeable in CSLL^∞ thanks to coexponentials, which enable the concurrent access to a shared resource. ⌟

We conclude this section by defining various termination properties of interest. A *run* of P is a (finite or infinite) sequence $(P_0, P_1, \dots)$ of processes such that $P = P_0$ and $P_i \to P_{i+1}$ whenever P_{i+1} is a term in the sequence. A run is *maximal* if it is infinite or if it is finite and its last term (say Q) cannot reduce any further (that is, $Q \not\to$). We say that P is *terminating* if every maximal run of P is finite. We say that P is *weakly terminating* if P has a maximal finite run. A run of P is *fair* if it contains finitely many weakly terminating processes. We say that P is *fairly terminating* if every fair run of P is finite. Note that a fairly terminating process may admit infinite runs, but these go through infinitely many weakly terminating states. In other words, these runs represent executions of the process in which termination is always within reach but also always avoided, as if the system or the process itself is conspiring against termination. For this reason, these runs are considered "uninteresting" as far as termination is concerned and the process is considered to be practically terminating.

A fundamental property of any fairness notion is the fact that every finite run of a process should be extendable to a maximal fair one. This property, called *feasibility* [1] or *machine closure* [18], holds for our fairness notion and follows immediately from the next proposition.

▶ **Proposition 2.** *Every process has at least one maximal fair run.*

Proof. For an arbitrary process P there are two possibilities. If P is weakly terminating, then there exists Q such that $P \Rightarrow Q \not\to$. From this sequence of reductions we obtain a maximal run of P that is fair since it is finite. If P is not weakly terminating, then $P \to$ and $P \Rightarrow Q$ implies that Q is not weakly terminating. In this case we can build an infinite run of P which is fair since it does not go through any weakly terminating process. ◀

The given notion of fair termination admits an alternative characterization that does not refer to fair runs. This characterization provides us with the key proof principle to show that well-typed CSLL^∞ processes fairly terminate (Section 5).

▶ **Theorem 3.** *P is fairly terminating iff $P \Rightarrow Q$ implies that Q is weakly terminating.*

Table 3 Typing rules for CSLL^∞.

$$\textsc{[call]}\ \frac{P \vdash \overline{x : T}}{\mathsf{A}\langle\overline{x}\rangle \vdash \overline{x : T}}\ \mathsf{A}(\overline{x}) \triangleq P \qquad \textsc{[cut]}\ \frac{P \vdash \Gamma, x : T \qquad Q \vdash \Delta, x : T^\perp}{(x)(P \mid Q) \vdash \Gamma, \Delta} \qquad [\top]\ \frac{}{\mathsf{fail}\, x \vdash \Gamma, x : \top}$$

$$\textbf{no rule for } \mathbf{0} \qquad [\bot]\ \frac{P \vdash \Gamma}{\mathsf{wait}\, x.P \vdash \Gamma, x : \bot} \qquad [\mathbf{1}]\ \frac{}{\mathsf{close}\, x \vdash x : \mathbf{1}} \qquad [\parr]\ \frac{P \vdash \Gamma, y : T, x : S}{x(y).P \vdash \Gamma, x : T \parr S}$$

$$[\otimes]\ \frac{P \vdash \Gamma, y : T \qquad Q \vdash \Delta, x : S}{x[y](P \mid Q) \vdash \Gamma, \Delta, x : T \otimes S} \qquad [\&]\ \frac{P \vdash \Gamma, x : T \qquad Q \vdash \Gamma, x : S}{\mathsf{case}\, x\{P, Q\} \vdash \Gamma, x : T \,\&\, S} \qquad [\oplus]\ \frac{P \vdash \Gamma, x : T_i}{\mathsf{in}_i\, x.P \vdash \Gamma, x : T_1 \oplus T_2}$$

$$\textsc{[server]}\ \frac{P \vdash \Gamma, x : ¡T, y : T \qquad Q \vdash \Gamma}{¡x(y)\{P, Q\} \vdash \Gamma, x : ¡T} \qquad \textsc{[client]}\ \frac{P \vdash \Gamma, y : T \qquad Q \vdash \Delta, x : ¿T}{¿x[y].P :: Q \vdash \Gamma, \Delta, x : ¿T} \qquad \textsc{[done]}\ \frac{}{¿x[] \vdash x : ¿T}$$

Proof. ($\Leftarrow$) Suppose by contradiction that $(P_0, P_1, \dots)$ is an infinite fair run of P and note that $P \Rightarrow P_i$ for every i. From the hypothesis we deduce that every P_i is weakly terminating. Then the run contains infinitely many weakly terminating processes, which is absurd by definition of fair run. ($\Rightarrow$) Suppose that $P \Rightarrow Q$. Then there is a finite run of P that ends in Q. By Proposition 2 there is a maximal fair run of Q. By concatenating these two runs we obtain a maximal fair run of P that contains Q. From the hypothesis we deduce that this run is finite. Since Q occurs in this run, we conclude that Q is weakly terminating. ◀

3 Type System

In this section we develop the type system of CSLL^∞. Types are defined thus:

$$\textbf{Type} \qquad T, S ::= \bot \mid \mathbf{1} \mid \top \mid \mathbf{0} \mid T \parr S \mid T \otimes S \mid T \,\&\, S \mid T \oplus S \mid ¡T \mid ¿T$$

Types extend the usual constants and connectives of multiplicative-additive linear logic with the coexponentials $¡T$ and $¿T$ and, in the context of CSLL^∞, they describe how channels are used by processes. Positive types indicate output operations whereas negative types indicate input operations. In particular: $\mathbf{1}/\bot$ describe a session channel used for sending/receiving a session termination signal; $\mathbf{0}/\top$ describe a session channel used for sending/receiving an impossible (empty) message; $T \otimes S / T \parr S$ describe a session channel used for sending/receiving a channel of type T and then according to S; $T_1 \oplus T_2 / T_1 \,\&\, T_2$ describe a session channel used for sending/receiving a label in_i and then according to T_i; finally, $¿T/¡T$ describe a shared channel used for sending/receiving a connection message establishing a session of type T. Each type T has a *dual* $T^\perp$ obtained in the expected way. For example, we have $(\mathbf{1} \oplus T)^\perp = \bot \,\&\, T^\perp$ and $(¡T)^\perp = ¿T^\perp$.

The typing rules for CSLL^∞ are shown in Table 3. Typing judgments have the form $P \vdash \Gamma$ and relate a process P with a *context* Γ. Contexts are finite maps from channel names to types written as $\overline{x : T}$. We let Γ and Δ range over contexts, we write $\emptyset$ for the empty context, we write $\mathsf{dom}(\Gamma)$ for the domain of Γ, namely for the set of channel names for which there is an association in Γ, and we write Γ, Δ for the union of Γ and Δ when $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) = \emptyset$.

For the most part, the typing rules coincide with those of a standard session type system based on linear logic [21, 19]. In particular, [$\textsc{cut}$], [$\top$], [$\bot$], [$\mathbf{1}$], [$\parr$], [$\otimes$], [$\&$] and [$\oplus$] relate the standard proof rules of multiplicative-additive classical linear logic with the corresponding forms of CSLL^∞. The rule [$\textsc{call}$] deals with process invocations $\mathsf{A}\langle\overline{x}\rangle$ by unfolding the global definition of A, noted as side condition to the rule. Rule [$\textsc{server}$] deals with servers $\text{¡}x(y)\{P,Q\}$. The continuation P, which is the actual handler of incoming connections, must be well typed in a context enriched with the channel y resulting from the connection. Note that x is still present in the context and with the same type, meaning that P must *also* be able to handle any further connection on the shared channel x. The continuation Q, which models the behavior of the server once no more clients are connecting on x, is not supposed to use x any longer. Rule [$\textsc{client}$] deals with non-empty client pools $\text{¿}x[y].P :: Q$. The client P is connecting with a server through a shared channel x and establishes a session y. The rest of the pool Q is using x in the same way. Rule [$\textsc{done}$] deals with the empty pool of clients connecting on x.[1]

The typing rules are interpreted coinductively. Therefore, a judgment $P \vdash \Gamma$ is derivable if there is a possibly infinite typing derivation for it. The need for infinite typing derivations stems from the fact that we type process invocations by "unfolding" them to the process they represent, so this unfolding may go on forever in the case of recursive processes.

▶ **Example 4.** Let us consider once again the process definitions in Example 1. We derive

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\mathsf{Lock}\langle x,z\rangle \vdash x : \text{¡}\bot, z : \mathbf{1}}\,[\textsc{call}]}{\mathsf{wait}\,y.\mathsf{Lock}\langle x,z\rangle \vdash x : \text{¡}\bot, y : \bot, z : \mathbf{1}}\,[\bot] \qquad \dfrac{}{\mathsf{close}\,z \vdash z : \mathbf{1}}\,[\mathbf{1}]}{\text{¡}x(y)\{\mathsf{wait}\,y.\mathsf{Lock}\langle x,z\rangle, \mathsf{close}\,z\} \vdash x : \text{¡}\bot, z : \mathbf{1}}\,[\textsc{server}]}{\mathsf{Lock}\langle x,z\rangle \vdash x : \text{¡}\bot, z : \mathbf{1}}\,[\textsc{call}]$$

showing that Lock is well typed. Note that the typing derivation is infinite since Lock is a recursive process. We can now obtain the following typing derivation

$$\dfrac{\dfrac{\dfrac{}{\mathsf{close}\,u \vdash u : \mathbf{1}}\,[\mathbf{1}] \qquad \dfrac{\dfrac{}{\mathsf{close}\,v \vdash v : \mathbf{1}}\,[\mathbf{1}] \qquad \dfrac{}{\text{¿}x[\,] \vdash x : \text{¿}\mathbf{1}}\,[\textsc{done}]}{\text{¿}x[v].\mathsf{close}\,v :: \text{¿}x[\,] \vdash x : \text{¿}\mathbf{1}}\,[\textsc{client}]}{\text{¿}x[u].\mathsf{close}\,u :: \text{¿}x[v].\mathsf{close}\,v :: \text{¿}x[\,] \vdash x : \text{¿}\mathbf{1}}\,[\textsc{client}] \qquad \dfrac{\vdots}{\mathsf{Lock}\langle x,z\rangle \vdash x : \text{¡}\bot, z : \mathbf{1}}}{(x)(\text{¿}x[u].\mathsf{close}\,u :: \text{¿}x[v].\mathsf{close}\,v :: \text{¿}x[\,] \mid \mathsf{Lock}\langle x,z\rangle) \vdash z : \mathbf{1}}\,[\textsc{cut}]$$

showing that the system as a whole is well typed. ⌟

Adopting an infinitary type system will make it easy to relate CSLL^∞ with $\mu\mathsf{MALL}^\infty$ (Section 5). However, we must be careful in that some infinite typing derivations allow us to type processes that are not weakly terminating, as illustrated in the next example.

▶ **Example 5** (non-terminating process)**.** Consider the process $\Omega \triangleq (x)(\mathsf{close}\,x \mid \mathsf{wait}\,x.\Omega)$ which creates a session x, immediately closes it and then repeats the same behavior. Clearly, this process is not weakly terminating because it can only reduce thus:

$$\Omega \preccurlyeq (x)(\mathsf{close}\,x \mid \mathsf{wait}\,x.\Omega) \rightarrow \Omega \preccurlyeq (x)(\mathsf{close}\,x \mid \mathsf{wait}\,x.\Omega) \rightarrow \cdots$$

[1] There are some analogies between the typing rules for client pools and *coweakening*, *codereliction* and *cocontraction* in Differential Linear Logic (DiLL) [9], although the exact relationship between coexponentials and DiLL remains to be established. Quian et al. [20] provide a few more details.

Nonetheless, we are able to find the following (infinite) typing derivation for Ω.

$$\dfrac{\dfrac{}{\mathsf{close}\,x \vdash x : \mathbf{1}}\,[\mathbf{1}] \quad \dfrac{\dfrac{\vdots}{\Omega \vdash \emptyset}\,[\text{CALL}]}{\mathsf{wait}\,x.\Omega \vdash x : \bot}\,[\bot]}{\dfrac{(x)(\mathsf{close}\,x \mid \mathsf{wait}\,x.\Omega) \vdash \emptyset}{\Omega \vdash \emptyset}\,[\text{CALL}]}\,[\text{CUT}]$$

Since we aim at ensuring fair termination for well-typed processes, we must consider this derivation as *invalid*. ⌟

In order to rule out processes like Ω in Example 5, we identify a class of valid typing derivations as follows.

▶ **Definition 6** (valid typing derivation). *A typing derivation is* valid *if every infinite branch in it goes through infinitely many applications of the rule* [SERVER] *concerning the same channel.*

This validity condition requires that every infinite branch of a typing derivation describes the behavior of a server willing to accept an unbounded number of connection requests. If we look back at the infinite typing derivation for the Lock process in Example 4, we see that it is valid according to Definition 6 since the only infinite branch in it goes through infinitely many applications of the rule [SERVER] concerning the very same shared channel x. On the contrary, the typing derivation in Example 5 is invalid since the infinite branch in it does not go through any application of [SERVER].

The fact that every infinite branch must go through infinitely many applications of [SERVER] *concerning the very same shared channel* is a subtle point. Without the specification that it is the *same* shared channel to be found infinitely often, it would be possible to obtain invalid typing derivations as illustrated by the next example.

▶ **Example 7.** Consider the definition

$$\Omega\text{-}\mathsf{Server}(x) \triangleq {¡}x(y)\{\mathsf{wait}\,y.\Omega\text{-}\mathsf{Server}\langle x\rangle, (z)({¿}z[] \mid \Omega\text{-}\mathsf{Server}\langle z\rangle)\}$$

describing a server that waits for connections on the shared channel x. After each request, the server makes itself available again for handling more requests by the recursive invocation $\Omega\text{-}\mathsf{Server}\langle x\rangle$. Once all requests have been processed, the server creates a new shared channel on which an analogous server operates. Using the typing rules in Table 3 we are able to find the following typing derivation:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Omega\text{-}\mathsf{Server}\langle x\rangle \vdash x : {¡}\bot}\,[\text{CALL}]}{\mathsf{wait}\,y.\Omega\text{-}\mathsf{Server}\langle x\rangle \vdash x : {¡}\bot, y : \bot}\,[\bot] \quad \dfrac{\dfrac{}{{¿}z[] \vdash z : {¿}\mathbf{1}}\,[\text{DONE}] \quad \dfrac{\vdots}{\Omega\text{-}\mathsf{Server}\langle z\rangle \vdash z : {¡}\bot}\,[\text{CALL}]}{(z)({¿}z[] \mid \Omega\text{-}\mathsf{Server}\langle z\rangle) \vdash \emptyset}\,[\text{CUT}]}{\dfrac{{¡}x(y)\{\mathsf{wait}\,y.\Omega\text{-}\mathsf{Server}\langle x\rangle, (z)({¿}z[] \mid \Omega\text{-}\mathsf{Server}\langle z\rangle)\} \vdash x : {¡}\bot}{\Omega\text{-}\mathsf{Server}\langle x\rangle \vdash x : {¡}\bot}\,[\text{CALL}]}\,[\text{SERVER}]$$

Notice that the derivation bifurcates in correspondence of the application of [SERVER] and also that each sub-tree is infinite, since it contains an unfolding of the Ω-Server process. For this reason, the derivation contains (infinitely) many infinite branches, which are obtained by either "going left" or "going right" each time [SERVER] is encountered. Each of these infinite branches goes through an application of [SERVER] infinitely many times, as requested by

Definition 6. Also, any such branch that "goes right" finitely many times eventually ends up going through infinitely many applications of [SERVER] that concern the same channel. In contrast, any branch that "goes right" infinitely many times keeps going through applications of [SERVER] concerning new shared channels created in correspondence of the application of [CUT]. In conclusion, this typing derivation is invalid and rightly so, or else the diverging process $(x)(¿x[] \mid \Omega\text{-}\mathsf{Server}\langle x\rangle)$ would be well typed in the empty context. ⌟

We conclude this section by stating two key properties of the type system, starting from the fact that typing is preserved by structural pre-congruence and reductions.

▶ **Theorem 8.** *Let $P \mathrel{\mathcal{R}} Q$ where $\mathcal{R} \in \{\preccurlyeq, \rightarrow\}$. Then $P \vdash \Gamma$ implies $Q \vdash \Gamma$.*

Also, processes that are well typed in a context of the form $x : \mathbf{1}$ are deadlock free.

▶ **Theorem 9** (deadlock freedom). *If $P \vdash x : \mathbf{1}$ then either $P \preccurlyeq \mathsf{close}\, x$ or $P \rightarrow_{\mathsf{det}}$.*

Note that Theorem 9 uses $\rightarrow_{\mathsf{det}}$ instead of $\rightarrow$ in order to state that P is able to reduce if it is not (structurally pre-congruent to) $\mathsf{close}\, x$. Recalling that $\rightarrow_{\mathsf{det}} \subseteq \rightarrow$, the deadlock freedom property ensured by Theorem 9 is slightly stronger than one would normally expect. This formulation will be necessary in Section 5 when proving the soundness of the type system. The proofs of Theorems 8 and 9 can be found in Appendix A.

▶ **Example 10** (forwarder). Most session calculi based on linear logic include a form $x \leftrightarrow y$ whose typing rule $x \leftrightarrow y \vdash x : T, y : T^\perp$ corresponds to the axiom $\vdash T, T^\perp$ of linear logic. The form $x \leftrightarrow y$ is usually interpreted as a forwarder between the channels x and y and it is useful for example to model the output of a free channel $x\langle y\rangle.P$ as the term $x[z](y \leftrightarrow z \mid P)$. In this example we show that there is no need to equip CSLL^∞ with a native form $x \leftrightarrow y$ since its behavior can be encoded as a well-typed CSLL^∞ process. To this aim, we define a family Link_T of process definitions by induction on T as follows

$$
\begin{aligned}
\mathsf{Link}_\perp(x, y) &\triangleq \mathsf{wait}\, x.\mathsf{close}\, y \\
\mathsf{Link}_\top(x, y) &\triangleq \mathsf{fail}\, x \\
\mathsf{Link}_{T \parr S}(x, y) &\triangleq x(u).y[v](\mathsf{Link}_T\langle u, v\rangle \mid \mathsf{Link}_S\langle x, y\rangle) \\
\mathsf{Link}_{T \& S}(x, y) &\triangleq \mathsf{case}\, x\{\mathsf{in}_1\, y.\mathsf{Link}_T\langle x, y\rangle, \mathsf{in}_2\, y.\mathsf{Link}_S\langle x, y\rangle\} \\
\mathsf{Link}_{¡T}(x, y) &\triangleq ¡x(u)\{¿y[v].\mathsf{Link}_T\langle u, v\rangle :: \mathsf{Link}_{¡T}\langle x, y\rangle, ¿y[]\}
\end{aligned}
$$

with the addition of the definitions $\mathsf{Link}_T(x, y) \triangleq \mathsf{Link}_{T^\perp}\langle y, x\rangle$ for the positive type constructors. It is easy to build a typing derivation for the judgment $\mathsf{Link}_T\langle x, y\rangle \vdash x : T, y : T^\perp$. Also, every infinite branch in such derivation eventually loops through an invocation of the form $\mathsf{Link}_{¡S}\langle u, v\rangle$, which goes through an application of [SERVER] concerning the channel u. So, the derivation of $\mathsf{Link}_T\langle x, y\rangle \vdash x : T, y : T^\perp$ is valid and the process $\mathsf{Link}_T\langle x, y\rangle$ is well typed. ⌟

4 A quick recollection of μMALL$^\infty$

In this section we recall the main elements of $\mu\mathsf{MALL}^\infty$ [8, 3, 2], the infinitary proof system of the multiplicative additive fragment of linear logic extended with least and greatest fixed points. The syntax of $\mu\mathsf{MALL}^\infty$ *pre-formulas* makes use of an infinite set of *propositional variables* ranged over by X and Y and is given by the grammar below:

Pre-formula $\quad \varphi, \psi ::= X \mid \perp \mid \top \mid \mathbf{0} \mid \mathbf{1} \mid \varphi \parr \psi \mid \varphi \otimes \psi \mid \varphi \& \psi \mid \varphi \oplus \psi \mid \nu X.\varphi \mid \mu X.\varphi$

Table 4 Proof rules of μMALL$^\infty$ [3, 8, 2].

$$\frac{\vdash \Sigma, F \qquad \vdash \Theta, F^\perp}{\vdash \Sigma, \Theta}\,[\textsc{cut}] \qquad \frac{}{\vdash \Sigma, \top}\,[\top] \qquad \frac{\vdash \Sigma}{\vdash \Sigma, \bot}\,[\bot] \qquad \frac{}{\vdash \mathbf{1}}\,[\mathbf{1}] \qquad \frac{\vdash \Sigma, F, G}{\vdash \Sigma, F \parr G}\,[\parr] \qquad \frac{\vdash \Sigma, F \qquad \vdash \Theta, G}{\vdash \Sigma, \Theta, F \otimes G}\,[\otimes]$$

$$\frac{\vdash \Sigma, F \qquad \vdash \Sigma, G}{\vdash \Sigma, F \mathbin{\&} G}\,[\&] \qquad \frac{\vdash \Sigma, F_i}{\vdash \Sigma, F_1 \oplus F_2}\,[\oplus] \qquad \frac{\vdash \Sigma, F\{\nu X.F/X\}}{\vdash \Sigma, \nu X.F}\,[\nu] \qquad \frac{\vdash \Sigma, F\{\mu X.F/X\}}{\vdash \Sigma, \mu X.F}\,[\mu]$$

The fixed point operators μ and ν are the binders of propositional variables and the notions of free and bound variables are defined accordingly. A μMALL$^\infty$ *formula* is a closed pre-formula. We write $\{\varphi/X\}$ for the capture-avoiding substitution of all free occurrences of X with φ and $\varphi^\perp$ for the *dual* of φ, which is the involution such that

$$X^\perp = X \qquad (\mu X.\varphi)^\perp = \nu X.\varphi^\perp \qquad (\nu X.\varphi)^\perp = \mu X.\varphi^\perp$$

among the other expected equations. Postulating that $X^\perp = X$ is not a problem since we will always dualize formulas, which do not contain free propositional variables.

We write $\preceq$ for the *subformula ordering*, that is the least partial order such that $\varphi \preceq \psi$ if φ is a subformula of ψ. For example, if $\varphi \stackrel{\text{def}}{=} \mu X.\nu Y.(X \oplus Y)$ and $\psi \stackrel{\text{def}}{=} \nu Y.(\varphi \oplus Y)$ we have $\varphi \preceq \psi$ and $\psi \not\preceq \varphi$. When Φ is a set of formulas, we write $\mathsf{min}\,\Phi$ for its $\preceq$-minimum formula if it is defined. Occasionally we let $\star$ stand for an arbitrary binary connective (one of $\oplus$, $\otimes$, $\&$, or $\parr$) and σ stand for an arbitrary fixed point operator (either μ or ν).

In μMALL$^\infty$ it is important to distinguish among different *occurrences* of the same formula in a proof derivation. To this aim, formulas are annotated with *addresses*. We assume an infinite set $\mathcal{A}$ of *atomic addresses*, $\mathcal{A}^\perp$ being the set of their duals such that $\mathcal{A} \cap \mathcal{A}^\perp = \emptyset$ and $\mathcal{A}^{\perp\perp} = \mathcal{A}$. We use a and b to range over elements of $\mathcal{A} \cup \mathcal{A}^\perp$. An *address* is a string aw where $w \in \{i, l, r\}^*$. The dual of an address is defined as $(aw)^\perp = a^\perp w$. We use α and β to range over addresses, we write $\sqsubseteq$ for the prefix relation on addresses and we say that α and β are *disjoint* if $\alpha \not\sqsubseteq \beta$ and $\beta \not\sqsubseteq \alpha$.

A *formula occurrence* (or simply occurrence) is a pair φ_α made of a formula φ and an address α. We use F and G to range over occurrences and we extend to occurrences several operations defined on formulas. In particular: we use logical connectives to compose occurrences so that $\varphi_{\alpha l} \star \psi_{\alpha r} \stackrel{\text{def}}{=} (\varphi \star \psi)_\alpha$ and $\sigma X.\varphi_{\alpha i} \stackrel{\text{def}}{=} (\sigma X.\varphi)_\alpha$; the dual of an occurrence is obtained by dualizing both its formula and its address, that is $(\varphi_\alpha)^\perp \stackrel{\text{def}}{=} \varphi^\perp_{\alpha^\perp}$; occurrence substitution preserves the address in the type within which the substitution occurs, but forgets the address of the occurrence being substituted, that is $\varphi_\alpha\{\psi_\beta/X\} \stackrel{\text{def}}{=} \varphi\{\psi/X\}_\alpha$.

We write $\overline{F}$ for the formula obtained by forgetting the address of F. Finally, we write $\rightsquigarrow$ for the least reflexive relation on types such that $F_1 \star F_2 \rightsquigarrow F_i$ and $\sigma X.F \rightsquigarrow F\{\sigma X.F/X\}$.

The proof rules of μMALL$^\infty$ are shown in Table 4, where Σ and Θ range over sets of occurrences written as $F_1, \dots, F_n$. The rules allow us to derive *sequents* of the form $\vdash \Sigma$ and are standard except for $[\nu]$, which *unfolds* a greatest fixed point just like $[\mu]$ does. Being an infinitary proof system, μMALL$^\infty$ rules are meant to be interpreted coinductively. That is, a sequent $\vdash \Sigma$ is derivable if there exists an *arbitrary* (finite or infinite) proof derivation whose conclusion is $\vdash \Sigma$. Without a *validity condition* on derivations, such proof system is notoriously unsound. μMALL$^\infty$'s validity condition requires every infinite branch of a derivation to be supported by the continuous unfolding of a greatest fixed point. In order to formalize this condition, we start by defining *threads*, which are sequences of occurrences.

▶ **Definition 11** (thread). *A* thread *of F is a (finite or infinite) sequence of occurrences $(F_0, F_1, \dots)$ such that $F_0 = F$ and $F_i \rightsquigarrow F_{i+1}$ whenever $i + 1$ is a valid index of the sequence.*

Hereafter we use t to range over threads. For example, if we consider $\varphi \stackrel{\text{def}}{=} \mu X.(X \oplus \mathbf{1})$, we have that $t \stackrel{\text{def}}{=} (\varphi_a, (\varphi \oplus \mathbf{1})_{ai}, \varphi_{ail}, \dots)$ is an infinite thread of φ_a.

Among all threads, we are interested in finding those in which a ν-formula is unfolded infinitely often. These threads, called ν-threads, are precisely defined thus:

▶ **Definition 12** (ν-thread). *Let $t = (F_0, F_1, \dots)$ be an infinite thread, let $\overline{t}$ be the corresponding sequence $(\overline{F_0}, \overline{F_1}, \dots)$ of formulas and let $\mathsf{inf}(t)$ be the set of elements of $\overline{t}$ that occur infinitely often in $\overline{t}$. We say that t is a* ν-thread *if $\min \mathsf{inf}(t)$ is defined and is a ν-formula.*

If we consider the infinite thread t above, we have $\mathsf{inf}(t) = \{\varphi, \varphi \oplus \mathbf{1}\}$ and $\min \mathsf{inf}(t) = \varphi$, so t is *not* a ν-thread because φ is not a ν-formula. Consider instead $\varphi \stackrel{\text{def}}{=} \nu X.\mu Y.(X \oplus Y)$ and $\psi \stackrel{\text{def}}{=} \mu Y.(\varphi \oplus Y)$ and observe that ψ is the "unfolding" of φ. Now $t_1 \stackrel{\text{def}}{=} (\varphi_a, \psi_{ai}, (\varphi \oplus \psi)_{aii}, \varphi_{aiil}, \dots)$ is a thread of φ_a such that $\mathsf{inf}(t_1) = \{\varphi, \psi, \varphi \oplus \psi\}$ and we have $\min \mathsf{inf}(t_1) = \varphi$ because $\varphi \preceq \psi$, so t_1 is a ν-thread. If, on the other hand, we consider the thread $t_2 \stackrel{\text{def}}{=} (\varphi_a, \psi_{ai}, (\varphi \oplus \psi)_{aii}, \psi_{aiir}, (\varphi \oplus \psi)_{aiiri}, \dots)$ such that $\mathsf{inf}(t_2) = \{\psi, \varphi \oplus \psi\}$ we have $\min \mathsf{inf}(t_2) = \psi$ because $\psi \preceq \varphi \oplus \psi$, so t_2 is not a ν-thread. Intuitively, the $\preceq$-minimum formula among those that occur infinitely often in a thread is the outermost fixed point operator that is being unfolded infinitely often. It is possible to show that this minimum formula is always well defined [8]. If such minimum formula is a greatest fixed point operator, then the thread is a ν-thread. Note that a ν-thread is necessarily infinite.

Now we proceed by identifying threads along branches of proof derivations. To this aim, we provide a precise definition of *branch*.

▶ **Definition 13** (branch). *A* branch *of a proof derivation is a sequence $(\vdash \Sigma_0, \vdash \Sigma_1, \dots)$ of sequents such that $\vdash \Sigma_0$ occurs somewhere in the derivation and $\vdash \Sigma_{i+1}$ is a premise of the rule application that derives $\vdash \Sigma_i$ whenever $i + 1$ is a valid index of the sequence.*

An infinite branch is valid if supported by a ν-thread that originates somewhere therein.

▶ **Definition 14.** *Let $\gamma = (\vdash \Sigma_0, \vdash \Sigma_1, \dots)$ be an infinite branch in a derivation. We say that γ is* valid *if there exists $I \subseteq \mathbb{N}$ such that $(F_i)_{i \in I}$ is a ν-thread and $F_i \in \Sigma_i$ for every $i \in I$.*

▶ **Definition 15.** *A $\mu\mathsf{MALL}^\infty$ derivation is* valid *if so are its infinite branches.*

5 Fair Termination of CSLL$^\infty$

In this section we prove that well-typed CSLL$^\infty$ processes fairly terminate. We do so by appealing to the alternative characterization of fair termination given by Theorem 3. Using that characterization and using the fact that typing is preserved by reductions (Theorem 8), it suffices to show that well-typed CSLL$^\infty$ processes *weakly* terminate. To do that, we encode a well-typed CSLL$^\infty$ process P into a (valid) μMALL$^\infty$ proof and we use the cut elimination property of μMALL$^\infty$ to argue that P has a finite maximal run.

Encoding of types

The encoding of CSLL$^\infty$ types into μMALL$^\infty$ formulas is the map $[\![\cdot]\!]$ defined by

$$[\![¿T]\!] = \mu X.(\mathbf{1} \oplus ([\![T]\!] \otimes X)) \qquad [\![¡T]\!] = \nu X.(\bot \mathbin{\&} ([\![T]\!] \mathbin{⅋} X)) \tag{2}$$

and extended homomorphically to all the other type constructors, which are in one-to-one correspondence with the connectives and constants of $\mu\mathsf{MALL}^\infty$. Notice that the image of the encoding is a relatively small subset of $\mu\mathsf{MALL}^\infty$ formulas in which different fixed point operators are never intertwined. Also notice that the encoding of the coexponentials does not follow exactly their expansion in Equation (1). Basically, we choose to interpret $¿T$ as a *list* of clients rather than as a *tree* of clients, following to the intuition that clients are queued when connecting to a server. The interpretation of ¡ follows as a consequence, as we want it to be the dual of the interpretation of ¿. Note that this interpretation of the coexponential modalities is the same used by Qian et al. [20].

Encoding of typing contexts

The next step is the encoding of CSLL^∞ contexts into $\mu\mathsf{MALL}^\infty$ sequents. Recall that a $\mu\mathsf{MALL}^\infty$ sequent is a set of occurrences and that an occurrence is a pair φ_α made of a formula φ and an address α. In order to associate addresses with formulas, we parametrize the encoding of CSLL^∞ contexts with an injective map σ from CSLL^∞ channels to addresses, since channels in (the domain of a) CSLL^∞ context uniquely identify the occurrence of a type (and thus of a formula). We write $x \mapsto \alpha$ for the singleton map that associates x with the address α and σ_1, σ_2 for the union of σ_1 and σ_2 when they have disjoint domains and codomains. Now, the encoding of a CSLL^∞ context is set of formulas defined by

$$[\![x_1 : T_1, \dots, x_n : T_n]\!]_{\sigma, x_1\mapsto\alpha_1,\dots,x_n\mapsto\alpha_n} \stackrel{\mathsf{def}}{=} [\![T_1]\!]_{\alpha_1}, \dots, [\![T_n]\!]_{\alpha_n}$$

Encoding of typing derivations

Just like for the encoding of CSLL^∞ contexts, also the encoding of typing derivations is parametrized by a map σ from CSLL^∞ channels to addresses. In addition, we also have to take into account the possibility that *restricted channels* are introduced in a CSLL^∞ context, which happens in the rule [CUT] of Table 3. The formula occurrence corresponding to the type of this newly introduced channel must have an address that is disjoint from that of any other occurrence. To guarantee this disjointness, we parametrize the encoding of CSLL^∞ derivations by an *infinite stream* ρ of pairwise distinct atomic addresses. Formally, ρ is an injective function $\mathbb{N} \to \mathcal{A}$. We write $a\rho$, $\mathsf{even}(\rho)$ and $\mathsf{odd}(\rho)$ for the streams defined by

$$(a\rho)(0) \stackrel{\mathsf{def}}{=} a \qquad (a\rho)(n+1) \stackrel{\mathsf{def}}{=} \rho(n) \qquad \mathsf{even}(\rho)(n) \stackrel{\mathsf{def}}{=} \rho(2n) \qquad \mathsf{odd}(\rho)(n) \stackrel{\mathsf{def}}{=} \rho(2n+1)$$

respectively. In words, $a\rho$ is the stream of atomic addresses that starts with a and continues as ρ whereas $\mathsf{even}(\rho)$ and $\mathsf{odd}(\rho)$ are the sub-streams of ρ consisting of addresses with an even (respectively, odd) index.

The encoding of a CSLL^∞ typing derivation is coinductively defined by a map $[\![\cdot]\!]^\rho_\sigma$ which we describe using the following notation. For every typing rule in Table 3

$$\dfrac{\mathcal{J}_1 \qquad \cdots \qquad \mathcal{J}_n}{\mathcal{J}}\,[\text{RULE}] \qquad \text{we write} \qquad \left[\!\!\left[\dfrac{\mathcal{J}_1 \qquad \cdots \qquad \mathcal{J}_n}{\mathcal{J}} \right]\!\!\right]^\rho_\sigma = \pi$$

meaning that π is the $\mu\mathsf{MALL}^\infty$ derivation resulting from the encoding of the CSLL^∞ derivation for the judgment $\mathcal{J}$ in which the last rule is an application of [RULE]. Within π there will be instances of the $[\![\mathcal{J}_i]\!]^{\rho_i}_{\sigma_i}$ for suitable σ_i and ρ_i standing for the encodings of the CSLL^∞ sub-derivations for the judgments $\mathcal{J}_i$ that we find as premises of [RULE].

There is a close correspondence between many CSLL^∞ typing rules and $\mu\mathsf{MALL}^\infty$ proof rules so we only detail a few interesting cases of the encoding, starting from the typing rules [$\otimes$] and [$⅋$]. A $\mu\mathsf{MALL}^\infty$ typing derivation ending with an application of these rules is encoded as follows:

$$\left[\!\!\left[\frac{P \vdash \Gamma, y : T \qquad Q \vdash \Delta, x : S}{x[y](P \mid Q) \vdash \Gamma, \Delta, x : T \otimes S}\right]\!\!\right]^{\rho}_{\sigma, x \mapsto \alpha} = \frac{[\![P \vdash \Gamma, y : T]\!]^{\mathsf{even}(\rho)}_{\sigma, y \mapsto \alpha l} \qquad [\![Q \vdash \Delta, x : S]\!]^{\mathsf{odd}(\rho)}_{\sigma, x \mapsto \alpha r}}{\vdash [\![\Gamma, \Delta]\!]_\sigma, [\![T \otimes S]\!]_\alpha}\,[\otimes]$$

$$\left[\!\!\left[\frac{P \vdash \Gamma, y : T, x : S}{x(y).P \vdash \Gamma, x : T ⅋ S}\right]\!\!\right]^{\rho}_{\sigma, x \mapsto \alpha} = \frac{[\![P \vdash \Gamma, y : T, x : S]\!]^{\rho}_{\sigma, y \mapsto \alpha l, x \mapsto \alpha r}}{\vdash [\![\Gamma]\!]_\sigma, [\![T ⅋ S]\!]_\alpha}\,[⅋]$$

Notice that the types $T \otimes S$ and $T ⅋ S$ associated with x in the conclusion of the rules are encoded into the occurrences $[\![T \otimes S]\!]_\alpha$ and $[\![T ⅋ S]\!]_\alpha$ where α is the address associated with x in $\sigma, x \mapsto \alpha$. This address is suitably updated in the encoding of the premises of the rules. In the case of [$\otimes$], the original stream ρ of atomic addresses is split into two disjoint streams in the encoding of the premises to ensure that no atomic address is used twice.

Every application of [CALL] is simply erased in the encoding:

$$\left[\!\!\left[\frac{P \vdash \overline{x : T}}{\mathsf{A}\langle \overline{x} \rangle \vdash \overline{x : T}}\right]\!\!\right]^{\rho}_{\sigma} = [\![P \vdash \overline{x : T}]\!]^{\rho}_{\sigma}$$

The validity of the CSLL^∞ typing derivation guarantees that there cannot be an infinite chain of process invocations in a well-typed process. A proof of this fact is given by Lemma 21 in Appendix A. For this reason, the encoding of CSLL^∞ derivations is well defined despite the fact that applications of [CALL] are erased.

Another case worth discussing is that of the rule [CUT], which is handled as follows:

$$\left[\!\!\left[\frac{P \vdash \Gamma, x : T \qquad Q \vdash \Delta, x : T^\perp}{(x)(P \mid Q) \vdash \Gamma, \Delta}\right]\!\!\right]^{a\rho}_{\sigma} = \frac{[\![P \vdash \Gamma, x : T]\!]^{\mathsf{even}(\rho)}_{\sigma, x \mapsto a} \qquad [\![Q \vdash \Delta, x : T^\perp]\!]^{\mathsf{odd}(\rho)}_{\sigma, x \mapsto a^\perp}}{\vdash [\![\Gamma, \Delta]\!]_\sigma}\,[\mathrm{CUT}]$$

The first address from the infinite stream $a\rho$, which is guaranteed to be distinct from any other address used so far and that will be used in the rest of the encoding, is associated with the newly introduced variable x. Similarly to the case of [$\otimes$], the tail of the stream is split in the encoding of the two premises of [CUT] so as to preserve this guarantee.

We now consider the applications of [DONE], [CLIENT] and [SERVER] which account for the most relevant part of the encoding. These rule applications are encoded by considering the interpretation of the co-exponentials in terms of least and greatest fixed points (Equation (2)) and then by applying the suitable $\mu\mathsf{MALL}^\infty$ proof rules ([μ] and [ν] in particular). We have

$$\left[\!\!\left[\frac{}{¿x[] \vdash x : ¿T}\right]\!\!\right]^{\rho}_{\sigma, x \mapsto \alpha} = \dfrac{\dfrac{\dfrac{}{\vdash \mathbf{1}_{\alpha i l}}\,[\mathbf{1}]}{\vdash [\![\mathbf{1} \oplus (T \otimes ¿T)]\!]_{\alpha i}}\,[\oplus]}{\vdash [\![¿T]\!]_\alpha}\,[\mu]$$

for the applications of [DONE] and

$$\left[\!\!\left[\frac{P \vdash \Gamma, y : T \qquad Q \vdash \Delta, x : ¿T}{¿x[y].P :: Q \vdash \Gamma, \Delta, x : ¿T} \right]\!\!\right]^{\rho}_{\sigma, x \mapsto \alpha} = \dfrac{\dfrac{\dfrac{[\![P \vdash \Gamma, y : T]\!]^{\mathsf{even}(\rho)}_{\sigma, y \mapsto \alpha irl} \qquad [\![Q \vdash \Delta, x : ¿T]\!]^{\mathsf{odd}(\rho)}_{\sigma, x \mapsto \alpha irr}}{\vdash [\![\Gamma, \Delta]\!]_\sigma, [\![T \otimes ¿T]\!]_{\alpha ir}}\,[\otimes]}{\vdash [\![\Gamma, \Delta]\!]_\sigma, [\![\mathbf{1} \oplus (T \otimes ¿T)]\!]_{\alpha i}}\,[\oplus]}{\vdash [\![\Gamma, \Delta]\!]_\sigma, [\![¿T]\!]_\alpha}\,[\mu]$$

for the applications of [CLIENT]. Finally, the applications of [SERVER] are encoded thus:

$$\left[\!\!\left[\frac{P \vdash \Gamma, x : ¡T, y : T \qquad Q \vdash \Gamma}{¡x(y)\{P, Q\} \vdash \Gamma, x : ¡T} \right]\!\!\right]^{\rho}_{\sigma, x \mapsto \alpha} = \dfrac{\dfrac{\dfrac{[\![Q \vdash \Gamma]\!]^{\rho}_{\sigma}}{\vdash [\![\Gamma]\!]_\sigma, \bot_{\alpha il}}\,[\bot] \qquad \dfrac{[\![P \vdash \Gamma, x : ¡T, y : T]\!]^{\rho}_{\sigma, x \mapsto \alpha irr, y \mapsto \alpha irl}}{\vdash [\![\Gamma]\!]_\sigma, [\![T \parr ¡T]\!]_{\alpha ir}}\,[\parr]}{\vdash [\![\Gamma]\!]_\sigma, [\![\bot \mathbin{\&} (T \parr ¡T)]\!]_{\alpha i}}\,[\&]}{\vdash [\![\Gamma]\!]_\sigma, [\![¡T]\!]_\alpha}\,[\nu]$$

Note that in this last case it is not necessary to split the stream ρ since the P and Q branches of the server are mutually exclusive (the reduction rules [R-CONNECT] and [R-DONE] pick one or the other branch, but not both). A similar thing happens in the encoding of the applications of [&], not shown here.

Validity of encoded typing derivations

Now that we have shown how every CSLL^∞ typing derivation is encoded into a $\mu\mathsf{MALL}^\infty$ derivation, we argue that the encoding preserves validity. More specifically, a valid CSLL^∞ typing derivation (Definition 6) is encoded into a valid $\mu\mathsf{MALL}^\infty$ derivation (Definition 15). To see that this is the case, first observe that there is a one-to-one correspondence between the infinite branches in the two derivations. From Definition 6 we know that every infinite branch in a CSLL^∞ derivation contains infinitely many applications of [SERVER] concerning the same shared channel x having type $¡T$ for some T. In the encoded derivation, this translates to the existence of a formula $[\![¡T]\!]$ that occurs infinitely often in the sequents making up this infinite branch. Now, suppose that the first occurrence of this formula is associated with some address α. From the encoding of [SERVER] we can then build the thread

$$t \stackrel{\mathsf{def}}{=} ([\![¡T]\!]_\alpha, [\![\bot \mathbin{\&} (T \parr ¡T)]\!]_{\alpha i}, [\![T \parr ¡T]\!]_{\alpha ir}, [\![¡T]\!]_{\alpha irr}, \dots)$$

which is *infinite*. Also note that $\mathsf{inf}(t) = \{[\![¡T]\!], [\![\bot \mathbin{\&} (T \parr ¡T)]\!], [\![T \parr ¡T]\!]\}$, that $\min \mathsf{inf}(t) = [\![¡T]\!]$, and that $[\![¡T]\!]$ is a ν-formula by Equation (2). In conclusion, t is a ν-thread (Definition 12) as required by the validity condition for $\mu\mathsf{MALL}^\infty$ pre-proofs (Definition 15).

Soundness of the type system

Now that we know how to obtain a $\mu\mathsf{MALL}^\infty$ proof from a well-typed CSLL^∞ process we observe that each reduction rule of $\mathsf{CSLL}^\infty_{\mathsf{det}}$ corresponds to *one or more* principal reductions in a $\mu\mathsf{MALL}^\infty$ proof [8, Figure 3.2]. In particular, the reductions [R-CLOSE], [R-COMM] and [R-CASE] correspond to *exactly one* principal reduction in $\mu\mathsf{MALL}^\infty$ (for $\mathbf{1}/\bot$, $\otimes/\parr$ and $\oplus/\&$

respectively), whereas [R-DONE] and [T-CLIENT] correspond to *three* subsequent principal reductions in $\mu\mathsf{MALL}^\infty$. For example, [R-CONNECT] corresponds to the principal reduction μ/ν followed by $\oplus/\&$ followed by $\otimes/\parr$. Using this correspondence between $\mathsf{CSLL}^\infty_{\mathsf{det}}$ and $\mu\mathsf{MALL}^\infty$, we can prove that every CSLL^∞ process that is well typed in a context of the form $x : \mathbf{1}$ is weakly terminating. Note that this correspondence holds for $\rightarrow_{\mathsf{det}}$ but *not* for $\rightarrow$ in general. However, since $\rightarrow_{\mathsf{det}} \subseteq \rightarrow$, this is enough to establish the weak termination of well-typed CSLL^∞ processes in the general case.

▶ **Theorem 16.** *If $P \vdash x : \mathbf{1}$ then P is weakly terminating.*

Proof. Let $a\rho$ be an infinite stream of pairwise distinct atomic addresses. Every deterministic reduction of P (that is, according to $\rightarrow_{\mathsf{det}}$) can be mimicked by one or more principal reductions in the $\mu\mathsf{MALL}^\infty$ proof $[\![P \vdash x : \mathbf{1}]\!]^\rho_{x\mapsto a}$. We know that $\mu\mathsf{MALL}^\infty$ enjoys cut elimination [8]. In particular, there cannot be an infinite sequence of principal reductions in a $\mu\mathsf{MALL}^\infty$ proof [8, Proposition 3.5]. It follows that there is no infinite sequence of deterministic reductions starting from P (using the $\mathsf{CSLL}^\infty_{\mathsf{det}}$ semantics), that is $P \Rightarrow_{\mathsf{det}} Q \nrightarrow_{\mathsf{det}}$ for some Q. From Theorem 8 we deduce $Q \vdash x : \mathbf{1}$ and from Theorem 9 we deduce $Q \preccurlyeq \mathsf{close}\, x$. We conclude $P \Rightarrow\preccurlyeq \mathsf{close}\, x \nrightarrow$. In other words, P is weakly terminating. ◀

▶ **Corollary 17.** *If $P \vdash x : \mathbf{1}$ then P is fairly terminating.*

Proof. Straightforward consequence of Theorems 3 and 16. ◀

6 Example: a Compare-and-Swap register

In this section we illustrate a more complex scenario of client-server interaction that hightlights not only the fact that the server handles connections sequentially in an unspecified order but also the fact that each connection may change the server's internal state and affect other connections. More specifically, we show a modeling of the Compare-and-Swap (CAS) register of Qian et al. [20] in CSLL^∞. A CAS register holds a boolean value true or false and is represented as a server that accepts connections from clients. Each client sends two boolean values to the server, an *expected value* and a *desired value*. If the expected value matches the content of the register, then the register is overwritten with the desired value. Otherwise, the register remains unchanged. We model boolean values as choices made in some session y. For instance, we can model the sending of true on y by the selection $\mathsf{in}_1\, y$ and the sending of false on y by the selection $\mathsf{in}_2\, y$. In fact, in this section we write true and false as aliases for the labels in_1 and in_2, respectively.

Below are two definitions for clients that differ for the expected and desired values they send to the CAS register:

$$\mathsf{Client}_{\mathsf{true},\mathsf{false}}(y) \triangleq \mathsf{true}\, y.\mathsf{false}\, y.\mathsf{close}\, y \qquad \mathsf{Client}_{\mathsf{false},\mathsf{true}}(y) \triangleq \mathsf{false}\, y.\mathsf{true}\, y.\mathsf{close}\, y$$

It is easy to see that both definitions are well typed. In particular, we can derive $\mathsf{Client}_{b,c}\langle y\rangle \vdash y : (\mathbf{1}\oplus\mathbf{1})\oplus(\mathbf{1}\oplus\mathbf{1})$ for every $b, c \in \{\mathsf{true}, \mathsf{false}\}$ with two applications of $[\oplus]$ and one application of $[\mathbf{1}]$. We combine two clients in a single pool as by the following definition

$$\mathsf{Clients}(x) \triangleq \text{¿}x[y].\mathsf{Client}_{\mathsf{true},\mathsf{false}}\langle y\rangle :: \text{¿}x[y].\mathsf{Client}_{\mathsf{false},\mathsf{true}}\langle y\rangle :: \text{¿}x[]$$

for which we derive $\mathsf{Clients}\langle x\rangle \vdash x : \text{¿}((\mathbf{1}\oplus\mathbf{1})\oplus(\mathbf{1}\oplus\mathbf{1}))$ using [CLIENT] and [DONE].

For the CAS server we provide two definitions $\mathsf{CAS}_{\mathsf{true}}$ and $\mathsf{CAS}_{\mathsf{false}}$ corresponding to the states in which the register holds the value true and false, respectively.

$$\begin{aligned}
\mathsf{CAS}_{\mathsf{true}}(x,z) \triangleq {}&¡x(y)\{\mathsf{case}\,y\{\mathsf{case}\,y\{\mathsf{wait}\,y.\mathsf{CAS}_{\mathsf{true}}\langle x,z\rangle, \mathsf{wait}\,y.\mathsf{CAS}_{\mathsf{false}}\langle x,z\rangle\},\\
&\qquad\qquad\quad \mathsf{case}\,y\{\mathsf{wait}\,y.\mathsf{CAS}_{\mathsf{true}}\langle x,z\rangle, \mathsf{wait}\,y.\mathsf{CAS}_{\mathsf{true}}\langle x,z\rangle\}\},\\
&\quad \mathsf{true}\,z.\mathsf{close}\,z\}\\
\mathsf{CAS}_{\mathsf{false}}(x,z) \triangleq {}&¡x(y)\{\mathsf{case}\,y\{\mathsf{case}\,y\{\mathsf{wait}\,y.\mathsf{CAS}_{\mathsf{false}}\langle x,z\rangle, \mathsf{wait}\,y.\mathsf{CAS}_{\mathsf{false}}\langle x,z\rangle\},\\
&\qquad\qquad\quad \mathsf{case}\,y\{\mathsf{wait}\,y.\mathsf{CAS}_{\mathsf{true}}\langle x,z\rangle, \mathsf{wait}\,y.\mathsf{CAS}_{\mathsf{false}}\langle x,z\rangle\}\},\\
&\quad \mathsf{false}\,z.\mathsf{close}\,z\}
\end{aligned}$$

The server in state $b \in \{\mathsf{true}, \mathsf{false}\}$ waits for connections on the shared channel x. If there is no client, the server sends b on z and terminates. If a client connects, then a session y is established. At this stage the server performs two input operations to receive the expected and desired values from the client. If the expected value does *not* match b, then the desired value is ignored and the server recursively invokes itself in the same state b. If the expected value matches b, then the server recursively invokes itself in a state that matches the client's desired value.

It is not difficult to obtain derivations for the judgments $\mathsf{CAS}_b\langle x,z\rangle \vdash x : ¡((\bot \mathbin{\&} \bot) \mathbin{\&} (\bot \mathbin{\&} \bot)), z : \mathbf{1} \oplus \mathbf{1}$ for every $b \in \{\mathsf{true}, \mathsf{false}\}$. These derivations are valid since every infinite branch in them goes through an application of [SERVER] concerning the channel x. In conclusion, the CAS server is well typed and so is the composition $(x)(\mathsf{Clients}\langle x\rangle \mid \mathsf{CAS}_{\mathsf{true}}\langle x,z\rangle)$.

Note that the process $(x)(\mathsf{Clients}\langle x\rangle \mid \mathsf{CAS}_{\mathsf{true}}\langle x,z\rangle)$ is not deterministic since it may reduce to either $\mathsf{true}\,z.\mathsf{close}\,z$ or $\mathsf{false}\,z.\mathsf{close}\,z$ depending on the order in which clients connect. Indeed, if $\mathsf{Client}_{\mathsf{true},\mathsf{false}}$ connects first, then the state of the register changes from true to false and then the connection with the second client changes it back from false to true. If, on the other hand, $\mathsf{Client}_{\mathsf{false},\mathsf{true}}$ connects first (because [S-POOL-COMM] is used), then the initial state of the register does not change and then it is changed from true to false when the client $\mathsf{Client}_{\mathsf{true},\mathsf{false}}$ finally connects.

7 Concluding Remarks

CSLL [20] is a non-deterministic session calculus based on linear logic in which servers handle multiple client requests sequentially. In this work we have targeted the problem of proving the termination of well-typed CSLL processes. To this aim, we have introduced CSLL^∞, a variant of CSLL closely related to $\mu\mathsf{MALL}^\infty$ [3, 8, 2], the infinitary proof system for multiplicative additive linear logic with fixed points. We have shown that well-typed CSLL^∞ processes are fairly terminating by encoding CSLL^∞ typing derivations into $\mu\mathsf{MALL}^\infty$ proofs and using the cut elimination property of $\mu\mathsf{MALL}^\infty$. Although fair termination is weaker than termination, it is strong enough to imply livelock freedom, which was one of the motivations for proving termination in the original CSLL work [20]. In our work, fair termination is termination under the fairness assumption that termination is not avoided forever (Theorem 3). However, inspection of our proof (Section 5) reveals that the fairness assumption can be substantially weakened: the fair termination in CSLL^∞ is reduced to the termination in $\mathsf{CSLL}^\infty_{\mathsf{det}}$, meaning that fair termination in CSLL^∞ is guaranteed if client requests are handled in order.

CSLL^∞ differs from the original CSLL in a few ways. In the interest of simplicity, we have chosen to omit constructs for modeling (pools of) sequential clients and replicated servers which are meant to be typed using the traditional exponential modalities. These features are orthogonal to the ones we are interested in and we think that they can be accommodated without substantial challenges following the same technical development illustrated in the present paper. In fact, the general support to fixed points in $\mu\mathsf{MALL}^\infty$ allows for this and other extensions, such as (co)recursive session types [19, 7]. Another difference is that CSLL^∞ adopts a reduction semantics that is completely ordinary for a process calculus. In particular, reductions are *not* allowed under prefixes, restrictions can*not* be moved beyond prefixes and (unrelated) prefixes can*not* be swapped. Nonetheless, we are able to relate the reduction semantics of CSLL^∞ with the cut reduction strategy of $\mu\mathsf{MALL}^\infty$ since $\mu\mathsf{MALL}^\infty$ proofs, which can be infinite, are reduced bottom-up. For this reason, we find that $\mu\mathsf{MALL}^\infty$ provides a natural logical foundation for session calculi based on linear logic.

Just like CSLL, also CSLL^∞ is related to SILL_S [4, 5] and HCP_ND [17], two session calculi based on linear logic that allow for races and non-determinism. In SILL_S, sessions can be *shared* among more than two communicating processes. Access to a shared session is regulated by means of explicit acquire/release actions that manifest themselves as special modalities in session types. The flexibility gained by session sharing may compromise deadlock freedom, which can be recovered by means of additional type structure [5]. HCP_ND uses bounded exponentials [12] to implement client/server interactions in which the amount of channel sharing is known (and bounded) in advance. Neither CSLL nor CSLL^∞ require such bounds. For example, the forwarder process $\mathsf{Link}_{\mathsf{i}T}\langle x, y\rangle$ in Example 10 would be ill typed in HCP_ND since the number of clients that may be willing to connect on x is not known *a priori*.

References

1 Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 189–198. ACM Press, 1987. doi:10.1145/41625.41642.

2 David Baelde, Amina Doumane, Denis Kuperberg, and Alexis Saurin. Bouncing threads for circular and non-wellfounded proofs. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS, 2022*, 2022. arXiv:2005.08257.

3 David Baelde, Amina Doumane, and Alexis Saurin. Infinitary proof theory: the multiplicative additive case. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 – September 1, 2016, Marseille, France*, volume 62 of *LIPIcs*, pages 42:1–42:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CSL.2016.42.

4 Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. *Proc. ACM Program. Lang.*, 1(ICFP):37:1–37:29, 2017. doi:10.1145/3110281.

5 Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest deadlock-freedom for shared session types. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 611–639. Springer, 2019. doi:10.1007/978-3-030-17184-1_22.

6 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Math. Struct. Comput. Sci.*, 26(3):367–423, 2016. doi:10.1017/S0960129514000218.

7 Luca Ciccone and Luca Padovani. An infinitary proof theory of linear logic ensuring fair termination in the linear π-calculus. In Bartek Klin, Slawomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*, volume 243 of *LIPIcs*, pages 36:1–36:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CONCUR.2022.36.

8 Amina Doumane. *On the infinitary proof theory of logics with fixed points. (Théorie de la démonstration infinitaire pour les logiques à points fixes).* PhD thesis, Paris Diderot University, France, 2017. URL: https://tel.archives-ouvertes.fr/tel-01676953.

9 Thomas Ehrhard. An introduction to differential linear logic: proof-nets, models and antiderivatives. *Math. Struct. Comput. Sci.*, 28(7):995–1060, 2018. doi:10.1017/S0960129516000372.

10 Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer, 1986. doi:10.1007/978-1-4612-4886-6.

11 Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In Hartmut Ehrig, Robert A. Kowalski, Giorgio Levi, and Ugo Montanari, editors, *TAPSOFT'87: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Pisa, Italy, March 23-27, 1987, Volume 2: Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Functional and Logic Programming and Specifications (CFLP)*, volume 250 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 1987. doi:10.1007/BFb0014972.

12 Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: A modular approach to polynomial-time computability. *Theor. Comput. Sci.*, 97(1):1–66, 1992. doi:10.1016/0304-3975(92)90386-T.

13 Orna Grumberg, Nissim Francez, and Shmuel Katz. Fair termination of communicating processes. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 254–265, New York, NY, USA, 1984. Association for Computing Machinery. doi:10.1145/800222.806752.

14 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

15 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

16 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.

17 Wen Kokke, J. Garrett Morris, and Philip Wadler. Towards races in linear logic. *Log. Methods Comput. Sci.*, 16(4), 2020. URL: https://lmcs.episciences.org/6979.

18 Leslie Lamport. Fairness and hyperfairness. *Distributed Comput.*, 13(4):239–245, 2000. doi:10.1007/PL00008921.

19 Sam Lindley and J. Garrett Morris. Talking bananas: structural recursion for session types. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 434–447. ACM, 2016. doi:10.1145/2951913.2951921.

20 Zesen Qian, G. A. Kavvos, and Lars Birkedal. Client-server sessions in linear logic. *Proc. ACM Program. Lang.*, 5(ICFP):1–31, 2021. doi:10.1145/3473567.

21 Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014. doi:10.1017/S095679681400001X.

A Supplement to Section 3

In the proofs of Lemmas 18 and 19 below we only focus on the *derivability* of the typing judgment a structural pre-congruence or a reduction, without worrying about the *validity* of the derivation. It is easy to see that validity is preserved since both structural pre-congruence and reductions either change a *finite* region of the typing derivation or *remove* an entire sub-tree of the derivation (as in the case of [r-case]). Either way, the fact that every infinite branch in the residual derivation satisfies the validity conditions (Definition 6) follows from the hypothesis that the initial typing derivation is valid.

▶ **Lemma 18.** *If $P \vdash \Gamma$ and $P \preccurlyeq Q$ then $Q \vdash \Gamma$.*

Proof. By induction on the derivation of $P \preccurlyeq Q$ and by cases on the last rule applied. The proof is standard, we only discuss [S-PAR-POOL] for illustration purposes. In this case $P = (z)(¿x[y].P_1 :: P_2 \mid P_3) \preccurlyeq ¿x[y].P_1 :: (z)(P_2 \mid P_3) = Q$ where $z \not\in \mathsf{fn}(¿x[y].P_1)$. From [CUT] we deduce $¿x[y].P_1 :: P_2 \vdash \Gamma_{12}, z : T$ and $P_3 \vdash \Gamma_3, z : T^\perp$ where $\Gamma = \Gamma_{12}, \Gamma_3$. From [CLIENT] and $z \not\in \mathsf{fn}(¿x[y].P_1)$ we deduce $P_1 \vdash \Gamma_1, y : S$ and $P_2 \vdash \Gamma_2, x : ¿S, z : T$ where $\Gamma_{12} = \Gamma_1, \Gamma_2, x : ¿S$. We derive $(z)(P_2 \mid P_3) \vdash \Gamma_2, \Gamma_3, x : ¿S$ with one application of [CUT]. We conclude $Q \vdash \Gamma$ with one application of [CLIENT]. ◀

▶ **Lemma 19.** *If $P \vdash \Gamma$ and $P \to Q$ then $Q \vdash \Gamma$.*

Proof. By induction on the derivation of $P \to Q$ and by cases on the last rule applied.

- [R-CLOSE] Then $P = (x)(\mathsf{close}\, x \mid \mathsf{wait}\, x.Q) \to Q$. From [CUT], [$\mathbf{1}$] and [$\perp$] we deduce $\mathsf{close}\, x \vdash x : \mathbf{1}$ and $\mathsf{wait}\, x.Q \vdash \Gamma, x : \perp$. From [$\perp$] we conclude $Q \vdash \Gamma$.
- [R-COMM] Then $P = (x)(x[y](P_1 \mid P_2) \mid x(y).P_3) \to (y)(P_1 \mid (x)(P_2 \mid P_3)) = Q$. From [CUT] we deduce $x[y](P_1 \mid P_2) \vdash \Gamma_{12}, x : T \otimes S$ and $x(y).P_3 \vdash \Gamma_3, x : T^\perp \mathbin{⅋} S^\perp$ where $\Gamma = \Gamma_{12}, \Gamma_3$. From [$\otimes$] we deduce $P_1 \vdash \Gamma_1, y : T$ and $P_2 \vdash \Gamma_2, x : S$ where $\Gamma_{12} = \Gamma_1, \Gamma_2$. From [$⅋$] we deduce $P_3 \vdash \Gamma_3, y : T^\perp, x : S^\perp$. We derive $(x)(P_2 \mid P_3) \vdash \Gamma_2, \Gamma_3, y : T^\perp$ with one application of [CUT]. We conclude $(y)(P_1 \mid (x)(P_2 \mid P_3)) \vdash \Gamma$ with one application of [CUT].
- [R-CASE] Then $P = (x)(\mathsf{in}_i\, x.R \mid \mathsf{case}\, x\{Q_1, Q_2\}) \to (x)(R \mid Q_i) = Q$. From [CUT], [$\oplus$] and [$\&$] we deduce $\mathsf{in}_i\, x.R \vdash \Gamma_1, x : T_1 \oplus T_2$ and $\mathsf{case}\, x\{Q_1, Q_2\} \vdash \Gamma_2, x : T_1^\perp \mathbin{\&} T_2^\perp$ where $\Gamma = \Gamma_1, \Gamma_2$. From [$\oplus$] we deduce $R \vdash \Gamma_1, x : T_i$. From [$\&$] we deduce $Q_i \vdash \Gamma_2, x : T_i^\perp$ for $i = 1, 2$. We conclude $(x)(R \mid Q_i) \vdash \Gamma$ with one application of [CUT].
- [R-CONNECT] Then $P = (x)(¿x[y].P_1 :: P_2 \mid ¡x(y)\{Q_1, Q_2\}) \to (y)(P_1 \mid (x)(P_2 \mid Q_1)) = Q$. From [CUT], [CLIENT] and [SERVER] we deduce $¿x[y].P_1 :: P_2 \vdash \Gamma_{12}, x : ¿T$ and $¡x(y)\{Q_1, Q_2\} \vdash \Delta, x : ¡T^\perp$ where $\Gamma = \Gamma_{12}, \Delta$. From [CLIENT] we deduce $P_1 \vdash \Gamma_1, y : T$ and $P_2 \vdash \Gamma_2, x : ¿T$ where $\Gamma_{12} = \Gamma_1, \Gamma_2$. From [SERVER] we deduce $Q_1 \vdash \Delta, x : ¡T^\perp, y : T^\perp$. We derive $(x)(P_2 \mid Q_1) \vdash \Gamma_2, \Delta, y : T^\perp$ with one application of [CUT]. We conclude $(y)(P_1 \mid (x)(P_2 \mid Q_1)) \vdash \Gamma$ with another application of [CUT].
- [R-DONE] Then $P = (x)(¿x[] \mid ¡x(y)\{R, Q\}) \to Q$. From [CUT], [DONE] and [SERVER] we deduce $¿x[] \vdash x : ¿T$ and $¡x(y)\{R, Q\} \vdash \Gamma, x : ¡T^\perp$. From [SERVER] we conclude $Q \vdash \Gamma$.
- [R-PAR] Then $P = (x)(P_1 \mid P_2) \to (x)(Q_1 \mid P_2) = Q$ where $P_1 \to Q_1$. From [CUT] we deduce $P_1 \vdash \Gamma_1, x : T$ and $P_2 \vdash \Gamma_2, x : T^\perp$ where $\Gamma = \Gamma_1, \Gamma_2$. Using the induction hypothesis we derive $Q_1 \vdash \Gamma_1, x : T$. We conclude $(x)(Q_1 \mid P_2) \vdash \Gamma$ with an application of [CUT].
- [R-POOL] Then $P = ¿x[y].P_1 :: P_2 \to ¿x[y].P_1 :: Q_2 = Q$ where $P_1 \to Q_2$. From [CLIENT] we deduce $P_1 \vdash \Gamma_1, y : T$ and $P_2 \vdash \Gamma_2, x : ¿T$ where $\Gamma = \Gamma_1, \Gamma_2, x : ¿T$. Using the induction hypothesis we derive $Q_2 \vdash \Gamma_2, x : ¿T$. We conclude $¿x[y].P_1 :: Q_2 \vdash \Gamma$ with an application of [CLIENT].
- [R-STRUCT] Using the induction hypothesis with two applications of Lemma 18. ◀

In order to prove deadlock freedom it is convenient to introduce *reduction contexts* to make it easy to refer to unguarded sub-terms of a process. A reduction context is basically a process with a single hole denoted by $[\,]$.

Reduction context $\qquad \mathcal{C}, \mathcal{D} \quad ::= \quad [\,] \quad | \quad (x)(\mathcal{C} \mid P) \quad | \quad (x)(P \mid \mathcal{C})$

Note that holes cannot occur in the tail of client pools, that is $¿x[y].P :: \mathcal{C}$ is *not* a reduction context even though the tail of a client pool may reduce by means of [R-POOL]. The point is that, in order to prove deadlock freedom, it is never necessary to reduce the tail of a client pool. Hereafter we write $\mathcal{C}[P]$ for the process obtained by replacing the hole in $\mathcal{C}$ with P. Note that this notion of replacement may capture some channels occurring free in P.

Before addressing deadlock freedom, we prove the following proximity lemma, showing that it is always possible to move a restriction close to a process in which the restricted channel occurs free.

▶ **Lemma 20.** *If $x \in \mathsf{fn}(P) \setminus (\mathsf{fn}(\mathcal{C}) \cup \mathsf{bn}(\mathcal{C}))$ then $(x)(\mathcal{C}[P] \mid Q) \preccurlyeq \mathcal{D}[(x)(P \mid Q)]$ for some $\mathcal{D}$.*

Proof. By induction on $\mathcal{C}$ and by cases on its shape. We do not detail symmetric cases and we assume, without loss of generality, that $\mathsf{fn}(Q) \cap \mathsf{bn}(\mathcal{C}) = \emptyset$.

- $\mathcal{C} = [\,]$. We conclude by taking $\mathcal{D} \stackrel{\text{def}}{=} [\,]$ and by reflexivity of $\preccurlyeq$.
- $\mathcal{C} = (y)(\mathcal{C}' \mid R)$. Then $x \in \mathsf{fn}(P) \setminus (\mathsf{fn}(\mathcal{C}') \cup \mathsf{bn}(\mathcal{C}') \cup \mathsf{fn}(R) \cup \{y\})$. We derive

$$
\begin{array}{rlll}
(x)(\mathcal{C}[P] \mid Q) & = (x)((y)(\mathcal{C}'[P] \mid R) \mid Q) & \text{by definition of } \mathcal{C} \\
& \preccurlyeq (x)(Q \mid (y)(\mathcal{C}'[P] \mid R)) & \text{by [S-PAR-COMM]} \\
& \preccurlyeq (y)((x)(Q \mid \mathcal{C}'[P]) \mid R) & \text{by [S-PAR-ASSOC] since } x \notin \mathsf{fn}(R),\ y \notin \mathsf{fn}(Q) \\
& \preccurlyeq (y)((x)(\mathcal{C}'[P] \mid Q) \mid R) & \text{by [S-PAR-COMM]} \\
& \preccurlyeq (y)(\mathcal{D}'[(x)(P \mid Q)] \mid R) & \text{by ind. hyp. for some } \mathcal{D}' \\
& = \mathcal{D}[(x)(P \mid Q)] & \text{by taking } \mathcal{D} \stackrel{\text{def}}{=} (y)(\mathcal{D}' \mid R)
\end{array}
$$
◀

The next auxiliary result proves that, in a well-typed process, a finite number of applications of [S-CALL] is always sufficient to unfold all of the process invocations occurring in it. To this aim, we introduce some more terminology on processes. We say that P is a *guard* if it is not a parallel composition or a process invocation. Note that every guard specifies a topmost action on some channel x. In this case, we say that P is an x-guard. We say that P is *unguarded* in Q if $Q = \mathcal{C}[P]$ for some $\mathcal{C}$. We say that P is *unfolded* if $P = \mathcal{C}[Q]$ implies that Q is not an invocation.

▶ **Lemma 21.** *If $P \vdash \Gamma$ then there exists an unfolded Q such that $P \preccurlyeq Q$.*

Proof. Let the *call depth* of P be the natural number $\mathsf{cd}(P)$ inductively defined as follows:

$$
\mathsf{cd}(P) = \begin{cases} 1 + \mathsf{cd}(Q) & \text{if } P = \mathsf{A}\langle \overline{x} \rangle \text{ and } \mathsf{A}(\overline{x}) \triangleq Q \\ 1 + \max\{\mathsf{cd}(P_1), \mathsf{cd}(P_2)\} & \text{if } P = (x)(P_1 \mid P_2) \\ 0 & \text{otherwise} \end{cases}
$$

Roughly, $\mathsf{cd}(P)$ is the maximum depth in the typing derivation of P where an unguarded guard is encountered. To see that $\mathsf{cd}(P)$ is well defined, recall that in every infinite branch of a valid typing derivation there are infinitely many applications of [SERVER] and that a process of the form $¡x(y)\{Q, R\}$ is a guard. Therefore, the value of $\mathsf{cd}(P)$ is only determined by the portion of P's derivation tree that stops at each occurrence of a guard. This portion is finite. The proof proceeds by induction on $\mathsf{cd}(P)$ and by cases on the shape of P. The desired Q is obtained by applying [S-CALL] each time an unguarded invocation is encountered and the induction guarantees that this rewriting is finite. ◀

▶ **Theorem 9** (deadlock freedom). *If $P \vdash x : \mathbf{1}$ then either $P \preccurlyeq \mathsf{close}\,x$ or $P \rightarrow_{\mathsf{det}}$.*

Proof. By Lemma 21 we may assume, without loss of generality, that P is unfolded. We want to show that there are two x-guards in P that can synchronize. To this aim, let $\mathsf{guards}(P)$ be inductively defined as

$$
\mathsf{guards}(P) = \begin{cases} \mathsf{guards}(P_1) + \mathsf{guards}(P_2) & \text{if } P = (x)(P_1 \mid P_2) \\ 1 & \text{otherwise} \end{cases}
$$

and let $\mathsf{channels}(P)$ be inductively defined as

$$\mathsf{channels}(P) = \begin{cases} 1 + \mathsf{channels}(P_1) + \mathsf{channels}(P_2) & \text{if } P = (x)(P_1 \mid P_2) \\ 0 & \text{otherwise} \end{cases}$$

In words, $\mathsf{guards}(P)$ counts the number of unguarded guards in P whereas $\mathsf{channels}(P)$ counts the number of unguarded restrictions in P. It is easy to prove that $\mathsf{guards}(P) > \mathsf{channels}(P)$. So, there must be at least one channel name x such that P contains two unguarded x-guards. That is, $P = \mathcal{C}[(x)(\mathcal{C}_1[P_1] \mid \mathcal{C}_2[P_2])]$ and both P_1 and P_2 are x-guards and $x \notin \mathsf{fn}(\mathcal{C}_i) \cup \mathsf{bn}(\mathcal{C}_i)$ for $i = 1, 2$. Then we derive

$$\begin{array}{rcll} P & = & \mathcal{C}[(x)(\mathcal{C}_1[P_1] \mid \mathcal{C}_2[P_2])] & \text{by definition of } P,\ P_1 \text{ and } P_2 \\ & \preccurlyeq & \mathcal{C}[\mathcal{D}_1[(x)(P_1 \mid \mathcal{C}_2[P_2])]] & \text{by Lemma 20 for some } \mathcal{D}_1 \\ & \preccurlyeq & \mathcal{C}[\mathcal{D}_1[(x)(\mathcal{C}_2[P_2] \mid P_1)]] & \text{by [}\textsc{s-par-comm}\text{]} \\ & \preccurlyeq & \mathcal{C}[\mathcal{D}_1[\mathcal{D}_2[(x)(P_2 \mid P_1)]]] & \text{by Lemma 20 for some } \mathcal{D}_2 \end{array}$$

Now we reason by cases on the shape of P_1 and P_2, knowing that they are x-guards and that they are well typed in contexts that contain the associations $x : T$ and $x : T^\perp$ for some T. If $P_2 = ¿x[y].Q$ then $P_1 = ¡x(y)\{Q_1, Q_2\}$ and P may reduce using [R-CONNECT]. The case in which $P_1 = ¿x[y].Q$ is symmetric and can be handled in a similar way with an additional application of [S-PAR-COMM]. The cases in which one of P_1 and P_2 is $¿x[]$ can be handled analogously, deducing that P may reduce using [R-DONE]. The only cases left are when neither P_1 nor P_2 is a client or $¿x[]$. Then, P_1 and P_2 must be x-guards beginning with dual actions which can synchronize using one of the rules [R-CLOSE], [R-COMM] or [R-CASE], possibly with the help of an application of [S-PAR-COMM]. ◀