Client-Server Sessions in Linear Logic

ZESEN QIAN, Aarhus University, Denmark
G. A. KAVVOS, University of Bristol, United Kingdom
LARS BIRKEDAL, Aarhus University, Denmark

We introduce coexponentials, a new set of modalities for Classical Linear Logic. As duals to exponentials, the coexponentials codify a distributed form of the structural rules of weakening and contraction. This makes them a suitable logical device for encapsulating the pattern of a server receiving requests from an arbitrary number of clients on a single channel. Guided by this intuition we formulate a system of session types based on Classical Linear Logic with coexponentials, which is suited to modelling client-server interactions. We also present a session-typed functional programming language for client-server programming, which we translate to our system of coexponentials.

CCS Concepts: • **Theory of computation → Linear logic**; • **Computing methodologies → Concurrent programming languages**; **Parallel programming languages**.

Additional Key Words and Phrases: session types, linear logic, propositions as sessions, Curry-Howard, π-calculus, coexponential modality, client-server architecture

ACM Reference Format:
Zesen Qian, G. A. Kavvos, and Lars Birkedal. 2021. Client-Server Sessions in Linear Logic. *Proc. ACM Program. Lang.* 5, ICFP, Article 62 (August 2021), 31 pages. https://doi.org/10.1145/3473567

1 INTRODUCTION

The programme of *session types* [Honda et al. 1998; Vasconcelos 2012] aims to formulate behavioural type systems that capture the notion of a *session*—a structured, concurrent interaction between communicating agents. Very little is usually assumed about these agents: their only shared resource is usually a set of *channels* through which they can send and receive messages. On the other hand, ever since its inception it has been clear that *linear logic* [Girard 1987] has a deep and mystifying relationship with concurrency. Abramsky [1994] argued that π-calculi [Milner et al. 1992] and linear logic should be in a Curry-Howard correspondence [Bellin and Scott 1994]. Consequently, one should be able to use formulas of linear logic as types that specify concurrent interactions, thereby constructing a system of session types that is logically motivated. Session types and linear types have recently undergone a swift rapprochement beginning with the work of Caires and Pfenning [Caires and Pfenning 2010; Caires et al. 2016].

Despite these advances, the π-calculi that have been developed for Linear Logic suffer from dire expressive poverty. The typable processes are free of deadlock and nondeterminism, at the price of being unable to model even benign forms of race. One striking omission is that it is difficult to write down a well-typed process that represents two distinct clients being served by a server listening on a single channel. The goal of the present paper is to introduce a logical device, namely the *strong*

62

Authors' addresses: Zesen Qian, Aarhus University, Denmark, zesen.qian@cs.au.dk; G. A. Kavvos, University of Bristol, United Kingdom, alex.kavvos@bristol.ac.uk; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

2475-1421/2021/8-ART62
https://doi.org/10.1145/3473567

coexponential modalities, that will allow us to give a linear type to this extremely common pattern of concurrent interaction.

1.1 The Problem

Caires and Pfenning [2010] proposed a Curry-Howard correspondence in which Intuitionistic Linear Logic is used as a type system for the π-calculus [Milner et al. 1992]. This correspondence allows one to interpret formulas of linear logic as *session types*, i.e., as specifications of disciplined communication over a named channel. A few years later Wadler [2014] extended this interpretation to *Classical Linear Logic (CLL)*. Wadler's system, which is called *Classical Processes (CP)*, perfectly corresponds to Girard's original one-sided sequent system for CLL [1987]. Its typing judgments are of the form $P \vdash \Gamma$, where P is a π-calculus process, and Γ is a list $x_1 : A_1, \ldots, x_n : A_n$ of name-session type pairs, with A_i a formula of Classical Linear Logic. The operational semantics of CP led Wadler to the following interpretation of the connectives.

$\otimes$	output	$⅋$	input
$\&$	offer a choice	$\oplus$	make a choice
$!$	server	$?$	client

We follow a convention by which the multiplicative connectives $\otimes$, $⅋$ associate to the right. Thus a type like $A \otimes B ⅋ C$ is $A \otimes (B ⅋ C)$ and can be read as: output a (channel of type) A, then input a (channel of type) B, and proceed as C.

While the interpretation of the first four connectives is intuitive, something seems to have gone awry with the exponentials [Wadler 2014, §3.4]. We claim that the computational behaviour of exponentials in CP does not in fact accommodate what we would think of as client-server interaction. To begin, we consider the following aspects to be the main characteristics of a client-server architecture [van Steen and Tanenbaum 2017, §§2.3, 3.4]:

(i) There is a *server process*, which repeatedly provides a service.
(ii) There is a *pool of client processes*, each of which requests the said service.
(iii) There is a unique *end point* at which the clients may issue their requests to the server.
(iv) The underlying network is *inherently unreliable*: clients may be served out-of-order, i.e., in a *nondeterministic* manner.

While Wadler's interpretation faithfully captures (i) and (iii), it does not immediately enable the representation of (ii). Because of its deterministic behaviour, CP is incapable of modelling (iv).

A CP term $S \vdash x : !A$ can indeed 'serve' sessions of type A over the channel x. However, the reading of a term $C \vdash y : ?A$ as a process which behaves as a *pool of clients* along channel y is not so crisp. Recall the three rules of ?, namely weakening, dereliction, and contraction. In CP:

$$\frac{Q \vdash \Gamma}{Q \vdash \Gamma, x : ?A}\ ?w \qquad \frac{Q \vdash \Gamma, y : A}{?x[y].\,Q \vdash \Gamma, x : ?A}\ ?d \qquad \frac{Q \vdash \Gamma, x : ?A, y : ?A}{Q[x/y] \vdash \Gamma, x : ?A}\ ?c$$

Wadler interprets these rules as *client formation*. Weakening stands for the empty case of a pool of no clients. Dereliction represents a single client following session A. Given that $Q[x/y]$ denotes the term obtained by renaming all free occurrences of y in Q to x, contraction enables the aggregation of two client pools: two sessions of type $?A$ can be collapsed into one.

We argue that, of those interpretations, only the one for dereliction is tenable. In the case of weakening, we see that at least one process is involved in the premise. Hence, the 'pool' formed has at least one client in it, albeit one that does not communicate with the server. Likewise, contraction does not combine different clients, but different sessions owned by the same client. Beginning with a single process $P \vdash x : A, y : A$ we can use dereliction twice followed by contraction to obtain $?w[x].\,?w[y].\,P \vdash w : ?A$. This process will ask for two channels that communicate with session

A. Nevertheless, the result is still a single process, and not a pool of clients. Dually, the type $!A$ merely connotes a *shared channel*: a non-linearized, non-session channel which is used to spawn an arbitrary number of new sessions, each one of type A [Caires and Pfenning 2010, §3].

More alarmingly, there is no way to combine two distinct processes $P \vdash z : A$ and $Q \vdash w : A$ into a single process $\mathsf{pool}(x; z.\, P, w.\, Q) \vdash x : ?A$ communicating along a shared channel. As a remedy, Wadler introduces the Mix rule:

$$\dfrac{P \vdash \Gamma \qquad Q \vdash \Delta}{P \mid Q \vdash \Gamma, \Delta}\ \textsc{Mix}$$

Mix was carefully considered for inclusion in Linear Logic, but was rejected [Girard 1987, §V.4]. Informally, it allows two completely independent, non-intercommunicating processes to run 'in parallel.' We may then use contraction to merge them into a single client pool:

$$\dfrac{\dfrac{\dfrac{P \vdash z : A}{?x[z].\, P \vdash x : ?A}\ ?d \qquad \dfrac{Q \vdash w : A}{?y[w].\, P \vdash y : ?A}\ ?d}{?x[z].\, P \mid ?y[w].\, Q \vdash x : ?A, y : ?A}\ \textsc{Mix}}{?x[z].\, P \mid ?x[w].\, Q \vdash x : ?A}\ ?c$$

The operational semantics of the Mix rule in CP are studied by Atkey et al. [2016]. To formulate them correctly one needs also to add the rule

$$\dfrac{}{\mathsf{stop} \vdash \cdot}\ \textsc{Mix0}$$

Mix0 has a flavour of inconsistency to it, but it is otherwise useful. On the technical level, it lets us show that the operational semantics, which adds a reaction $P \mid Q \longrightarrow P' \mid Q$ whenever $P \longrightarrow P'$, is well-behaved (terminating, deadlock-free, and deterministic). In terms of computational interpretation, Mix0 represents a stopped process. This solves the second problem we pointed out above, viz. the formation of a vacuously empty client pool:

$$\dfrac{\dfrac{}{\mathsf{stop} \vdash \cdot}\ \textsc{Mix0}}{\mathsf{stop} \vdash x : ?A}\ ?w$$

Nevertheless, Mix and Mix0 are unbecoming rules. To begin, they are respectively equivalent to $\bot \multimap \mathbf{1}$ and $\mathbf{1} \multimap \bot$, and thereby conflate the two units. Moreover, it is well-known [Abramsky et al. 1996; Atkey et al. 2016; Bellin 1997; Girard 1987; Wadler 2014] that Mix is equivalent to

$$A \otimes B \multimap A \parr B \tag{$*$}$$

where $C \multimap D \stackrel{\text{def}}{=} C^{\perp} \parr D$.

Admitting this implication is unwise. At first glance, ($*$) merely weakens the separation between these connectives, and hence damages the interpretation of $\parr$ as input, and $\otimes$ as output. However, we argue that deeper problems lurk just beneath the surface. Abramsky et al. [1996, §3.4.2] describe a perspective on CLL which reads $A \parr B$ as *connected concurrency* (information *necessarily* flows between A and B [Girard 1987, §V.4]) and $A \otimes B$ as *disjoint concurrency* (no information flow between A and B whatsoever). The implication ($*$) makes $\otimes$ a special case of $\parr$. Hence, flow between the components of $A \otimes B$ is *permitted, but not obligatory* [Abramsky and Jagadeesan 1994, §3.2]. Thus, ($*$) allows us to *pretend* that there is flow of information between two clients.[1]

Nevertheless, generating the actual flow of information is seemingly impossible. Using Mix we can put together two clients $C_i \vdash c_i : A$, and get a single process $C_0 \mid C_1 \vdash c_0 : A, c_1 : A$. As the comma stands for $⅋$, we can only cut this with a server $S \vdash s : A^\perp \otimes A^\perp$. But, by the interpretation of $\otimes$ as disjoint concurrency, we see that the two client sessions will be served by disjoint server components. In other words, the server will *not* allow information to flow between clients, which does not conform to our usual conception of a stateful server! To enable this kind of flow, a server must use $⅋$. As we cannot cut a $⅋$ (in the server) with another $⅋$ (in the client pool), we are compelled to also accept the converse implication $A ⅋ B \multimap A \otimes B$ in order to convert one of the two $⅋$'s to $\otimes$. This forces $\otimes = ⅋$, which inescapably leads to deadlock [Atkey et al. 2016, §4.2].

Requiring $\otimes = ⅋$, a.k.a. *compact closure* [Abramsky et al. 1996; Barr 1991], is often deemed necessary for concurrency. In fact, Atkey et al. [2016] argue that this *conflation of dual connectives* ($\mathbf{1} = \perp$, $\otimes = ⅋$, and so on) is the source of all concurrency in Linear Logic. The objective of this paper is to argue that there is another way: we aim to augment the Caires-Pfenning interpretation of propositions-as-sessions with a certain degree of concurrency *without adding Mix*. We also wish to introduce just enough nondeterminism to convincingly model client-server interactions in a style that satisfies points (i)–(iv).

We shall achieve both of these goals with the introduction of *coexponentials*.[2]

1.2 Roadmap

First, in §2 we discuss the expression of the usual exponential modalities of linear logic (!?) as least and greatest fixed points. This leads us to a different definition of !, which we call the *strong exponential*. By taking a 'multiplicative dual' of these fixed point expressions, we reach two novel modalities, the *strong coexponentials*, for which we write ¡ and ¿. We refine coexponentials back into a weak form that is similar to the usual exponentials, and show that they coincide with weak exponentials in the presence of Mix and the *Binary Cut* rule.

Following that, in §3 we introduce a process calculus with strong coexponentials, which we call CSLL (Client-Server Linear Logic). This new system is in the style of Kokke et al. [2019a], which replaces the one-sided sequents with *hypersequents*. It is argued that coexponentials enable the collection of an arbitrary number of clients following session A into a *client pool*, which communicates on a channel that follows session $¿A$. Conversely, the rules for ¡ express the formation of a *server*, which can be cut with a client pool to serve its requests.

In §4 we present an extended example that illustrates the computational behaviour of coexponentials, namely an implementation of the *Compare-and-Set (CAS)* synchronization primitive. Our system neatly encapsulates the racy yet atomic behaviour implicit in such operations.

In §5 we explore the implications of coexponentials in a session-typed functional language. We extend Wadler's GV with constructs for client-server interaction, and translate them to coexponentials in CSLL. We take advantage of the higher-level notation to give several examples that would be tedious to program directly in CSLL.

We survey related work in §6, and make some concluding remarks in §7.

[1]This is evident in the Abramsky-Jagadeesan game semantics for MLL+MIX: a play in $A \otimes B$ projects to plays for A and B, but the Opponent can switch components at will. The fully complete model consists of *history-free* strategies, so there can only be non-stateful Opponent-mediated flow of information between A and B.

[2]The word 'coexponential' was used in Lafont and Streicher [1991, §6.4] to refer the ? connective.

2 EXPONENTIALS, FIXED POINTS, AND COEXPONENTIALS

2.1 Exponentials as Fixed Points

The exponential ('of course') modality of linear logic ! is used to mark a replicable formula. While describing a combinatory presentation of linear logic, Girard and Lafont [1987, §3.2] noticed that $!A$ can potentially be expressed as the fixed point

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

The three additive conjuncts on the RHS correspond to the three rules of the dual connective ?, namely weakening, dereliction, and contraction. As $\&$ is a *negative* connective, the choice of conjunct rests on the 'user' of the formula,[3] who may pick one of the three conjuncts at will.

One may thus be led to believe that, were we to allow fixed points for all *functors*, we could obtain $!A$ as the *fixed point* of a functor. Baelde [2012, §2.3] discusses this in the context of a system of higher-order CLL with least and greatest fixed points. Using the functors

$$F_A(\mathcal{X}) \stackrel{\text{def}}{=} \mathbf{1} \mathbin{\&} A \mathbin{\&} (\mathcal{X} \otimes \mathcal{X}) \qquad\qquad G_A(\mathcal{X}) \stackrel{\text{def}}{=} \bot \oplus A \oplus (\mathcal{X} \parr \mathcal{X})$$

one defines

$$!A \stackrel{\text{def}}{=} \nu F_A \qquad\qquad ?A \stackrel{\text{def}}{=} \mu G_A$$

where μ and ν stand for the least and greatest fixed point respectively. Just by expanding the fixed point rules, one then obtains certain derivable rules. While those for ? are the usual ones—weakening, dereliction, and contraction—the rule for ! is radically different:

$$\textsc{StrongExp}\ \frac{\vdash \Gamma, B \qquad \vdash B^{\perp}, \mathbf{1} \qquad \vdash B^{\perp}, A \qquad \vdash B^{\perp}, B \otimes B}{\vdash \Gamma, !A}$$

As foreshadowed by the use of a greatest fixed point, this rule is *coinductive*. To prove $!A$ from context Γ one must use it to construct a 'seed' value (or 'invariant') of type B. Moreover, this value must be discardable ($\vdash B^{\perp}, \mathbf{1}$), derelictable ($\vdash B^{\perp}, A$), and copyable ($\vdash B^{\perp}, B \otimes B$). This is eerily reminiscent of the *free commutative comonoids* used to build certain categorical models of Linear Logic [Melliès 2009, §7.2]. Because of the arbitrary choice of 'seed' type B, the system using this rule does not produce good behaviour under cut elimination: the normal forms do not satisfy the *subformula property* [Baelde 2012, §3]: not all detours are eliminated. We call the modality introduced by StrongExp the *strong exponential*.

Baelde shows that the standard ! rule can be derived from StrongExp. But while the strong exponential can simulate the standard exponential, it also enables a host of other computational behaviours under cut elimination. Put simply, the standard exponential ensures *uniformity*: each dereliction of $!A$ into an A must be reduced to the very same proof of A every time. This makes sense in at least two ways. First, when we embed intuitionistic logic into linear logic through the Girard translation, we expect that in a proof of $(A \to B)^o \stackrel{\text{def}}{=} !A^o \multimap B^o$ each use of the antecedent $!A$ produces the same proof of A. Second, we know that one way to construct the exponential in many 'degenerate' models of linear logic [Barr 1991; Melliès et al. 2018] is through the formula

$$!A \stackrel{\text{def}}{=} \mathop{\Large\&}_{n \in \mathbb{N}} A^{\otimes n} / {\sim_n}$$

where $A^{\otimes n} \stackrel{\text{def}}{=} A \otimes \cdots \otimes A$, and $A^{\otimes n} / {\sim_n}$ stands for the equalizer of $A^{\otimes n}$ under its $n!$ symmetries. Decoding the categorical language, this means that we take one $\&$ component for each multiplicity n, and each component consists of exactly n copies of the same proof of A.

[3] Also known as *external choice*. In the language of game semantics, the *opponent*.

In contrast, the ! rules derived from their fixed point presentation merely create an infinite tree of occurrences of A, and not all of them need be proven in the same way.

2.2 Deriving Coexponentials

Both exponentials (qua fixed points) are given by a tree where each fork is marked with a connective ($\otimes$ for !, $\parr$ for ?). The leaves of the tree are either marked with A, or with the corresponding unit. Turning this process on its head leads to two dual modalities, which we call the *coexponentials*.

More concretely, we define two functors by dualising the connective that adorns forks. We must not forget to change the units accordingly: we swap $\mathbf{1}$ (the unit for $\otimes$) with $\bot$ (the unit for $\parr$). Let

$$H_A(\mathcal{X}) \stackrel{\text{def}}{=} \bot \mathbin{\&} A \mathbin{\&} (\mathcal{X} \parr \mathcal{X}) \qquad\qquad K_A(\mathcal{X}) \stackrel{\text{def}}{=} \mathbf{1} \oplus A \oplus (\mathcal{X} \otimes \mathcal{X})$$

The strong coexponentials are then defined by

$$¡A \stackrel{\text{def}}{=} \nu H_A \qquad\qquad ¿A \stackrel{\text{def}}{=} \mu K_A$$

We define $(¿A)^\perp \stackrel{\text{def}}{=} ¡A^\perp$, and vice versa. This gives the following derived rules.

$$\frac{}{\vdash ¿A}\,¿w \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, ¿A}\,¿d \qquad \frac{\vdash \Gamma, ¿A \qquad \vdash \Delta, ¿A}{\vdash \Gamma, \Delta, ¿A}\,¿c$$

$$\frac{\vdash \Gamma, B \qquad \vdash B^\perp, \bot \qquad \vdash B^\perp, A \qquad \vdash B^\perp, B \parr B}{\vdash \Gamma, ¡A}\,¡$$

The rules for ¿ are *distributed forms* of the structural rules, while the ¡ rule gives a *strong coexponential*, analogous to the strong version of ! described in the previous section. The corresponding 'weak' coexponential is given by replacing the above ¡ rule with

$$\frac{\vdash \bigotimes ¿\Gamma, A}{\vdash \bigotimes ¿\Gamma, ¡A}\,¡$$

$¿\Gamma$ stands for the context obtained by applying ¿ to every formula in Γ, and $\bigotimes$ folds this context with a tensor. Unfortunately, the presence of this folding operation means that this rule is not well-behaved in proof-theoretic terms.

2.3 Exponentials vs. Coexponentials under Mix and Binary Cuts

In fact, we can show that, in the presence of additional rules, (weak) exponentials and (weak) coexponentials are interderivable up to provability. This is not merely a theoretical result: it demonstrates that, under the bonnet, Wadler's use of Mix for the formation of a client pool (which we sketched in §1.1) secretly introduces the coexponential modalities proposed here.

The requisite rules are Mix, and one of the *binary cut* or *multicut* rules:

$$\textsc{BiCut}\ \frac{\vdash \Gamma, A, B \qquad \vdash \Delta, A^\perp, B^\perp}{\vdash \Gamma, \Delta} \qquad\qquad \textsc{MultiCut}\ \frac{\vdash \Gamma, A_1, \ldots, A_n \qquad \vdash \Delta, {A_1}^\perp, \ldots, {A_n}^\perp}{\vdash \Gamma, \Delta}$$

BiCut cuts two formulas at once, and MultiCut an arbitrary number. These rules were first proposed in the context of Linear Logic by Abramsky [1993b] in the compact setting ($\otimes = \parr$). They are logically equivalent, but only the second one satisfies cut elimination [Atkey et al. 2016, §4.2]. We recall some folklore facts regarding the interderivability of certain formulas and Mix-like inference rules. Recall that $C \multimap D \stackrel{\text{def}}{=} C^\perp \parr D$. Some form of the following lemma may be found across the relevant literature [Abramsky et al. 1996; Atkey et al. 2016; Bellin 1997; Girard 1987; Wadler 2014].

Lemma 2.1. *The following rules are logically interderivable.*

(i) The axiom $\mathbf{1} \multimap \bot$ and the Mix0 *rule.*
(ii) The axiom $\bot \multimap \mathbf{1}$ and the Mix *rule.*
(iii) The axiom $A \otimes B \multimap A \parr B$ and the Mix *rule.*
(iv) The axiom $A \parr B \multimap A \otimes B$ and the BiCut *rule.*
(v) BiCut *and* MultiCut.

Moreover, Mix0 *is derivable from the axiom rule $\vdash A^{\bot}, A$ and* BiCut.

Armed with this, we can prove that:

Theorem 2.2. *In CLL with* Mix *and* BiCut*, exponentials and coexponentials coincide up to provability. That is: if we replace ? and ! in the rules for the exponentials with ¿ and ¡ respectively, the resultant rule is provable using the coexponential rules, and vice versa.*

This theorem confirms that exponentials and coexponentials are indeed symmetric with respect to multiplicativity. It also explains why exponentials can represent client-server interactions after introducing Mix [Kokke et al. 2019a; Wadler 2014]. Finally, the theorem extends to strong exponentials vs. strong coexponentials; the proof there is even simpler: under Mix and BiCut we have $\otimes = \parr$, so F_A, H_A and G_A, K_A are pairwise logically equivalent.

3 PROCESSES

In the rest of the paper we will argue that the logical observations we made in §2 have a computational interpretation as client-server interaction. To this end we will introduce a process calculus for CLL equipped with a bespoke form of strong coexponentials. Our system shall introduce a certain amount of nondeterminism, yet it will remain Mix-free.

We first explain how the coexponentials capture the intuitive shape of client pool formation (§3.1). Following that, we briefly discuss three technical design decisions that pertain to the coexponentials used in our system (§§3.2–3.4). Finally, we introduce the system in §3.5, and its metatheory in §3.6.

3.1 ¿ Means Client, ¡ Means Server

Recall the three rules for ¿, namely

$$\frac{}{\vdash ¿A}\ ¿w \qquad\qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, ¿A}\ ¿d \qquad\qquad \frac{\vdash \Gamma, ¿A \qquad \vdash \Delta, ¿A}{\vdash \Gamma, \Delta, ¿A}\ ¿c$$

We can read $¿A$ as the session type of a channel shared by a *pool of clients.*

- $¿w$ allows the vacuous formation of a empty client pool.
- $¿d$ allows the formation of a client pool consisting of exactly one client.
- $¿c$ can be used to aggregate two client pools together.

The last point requires some elaboration. Each premise of $¿c$ can be seen as a client pool with an external interface (Γ and Δ respectively). The rule allows us to combine these into a single process. This new process still behaves as a client pool, but it also retains *both* external interfaces. In contrast, the $?c$ rule only allowed us to collapse two shared channels that belonged to *a single process.* Moreover, it did not allow us to mix two external interfaces—one had to use Mix for that.

Finally, the 'weak' ¡ rule, i.e.,

$$\frac{\vdash \bigotimes ¿\Gamma, A}{\vdash \bigotimes ¿\Gamma, ¡A}$$

can be read as the introduction rule for a dual *server session type*. It states that a process serving A, and all of whose other interactions have a client role (¿) with respect to a set of non-interacting ($\otimes$) services, can itself be 'co-promoted' to a *server* ¡A.

Note that our intuitive explanations are almost identical to those of Wadler [2014]; the difference is that our rules have the right branching structure to support the underlying intuition.

3.2 Design Decision #1: Server State and The Strong Rules

The first change with respect to the above is the switch to the *strong rule*, namely

$$\frac{\vdash \Gamma, B \qquad \vdash B^\perp, \perp \qquad \vdash B^\perp, A \qquad \vdash B^\perp, B \parr B}{\vdash \Gamma, ¡A}$$

This rule evokes the structure of a 'stateful' server serving A's, with external interface Γ. Within the server there exists an *internal server protocol* B. This comes with four ingredients: a process that provides a B, interacting along Γ (initialization); a way to silently consume B (finalization); a way to 'convert' a B to an A (serving a client); and a way to fork one B into two connected B's (forking two subservers).

We use this strong rule in order to avoid the *uniformity* property that was discussed in §2.1: the weak coexponential rule gives trivial servers providing identical A's to all clients. In contrast, this rule will allow a server to provide a different A each time it is called upon to do so.

3.3 Design Decision #2: Replacing Trees with Lists

The strong coexponential rule arose by taking the greatest fixed point of

$$H_A(\mathcal{X}) \stackrel{\text{def}}{=} \perp \,\&\, A \,\&\, (\mathcal{X} \parr \mathcal{X})$$

As discussed in §§2.1 and 2.2, this rule represents a *tree-like structure*. Nothing stops us from replacing it with a *list-like* structure.[4] We use the functors

$$H'_A(\mathcal{X}) \stackrel{\text{def}}{=} \perp \,\&\, (A \parr \mathcal{X}) \qquad\qquad K'_A(\mathcal{X}) \stackrel{\text{def}}{=} \mathbf{1} \oplus (A \otimes \mathcal{X})$$

and acquire the strong server rule derived from H'_A, viz.

$$\frac{\vdash \Gamma, B \qquad \vdash B^\perp, \perp \qquad \vdash B^\perp, A \parr B}{\vdash \Gamma, ¡A}$$

The main benefit is that the resulting system more closely reflects the pattern of client-server interaction: clients form a queue rather than a tree, and servers no longer have to fork subprocesses. This rule also requires fewer ingredients: an initialization of the internal protocol, a finalization, and a component that spawns a session to serve one additional client.

To optimize this further, we make the $\parr$ implicit, and replace $\perp$ with a general Δ in the finalization:

$$\frac{\vdash \Gamma, B \qquad \vdash B^\perp, \Delta \qquad \vdash B^\perp, A, B}{\vdash \Gamma, \Delta, ¡A}\ \textsc{Server}$$

[4]It is worth noting that Girard considered list-like exponentials [1987, §V.5(ii)], but rejected them as they were not able to reproduce contraction. This is not a requirement for modelling client-server interaction.

This second rule can be immediately derived from the first one:

$$\dfrac{\dfrac{\vdash \Gamma, B \qquad \vdash B^\perp, \Delta}{\vdash \Gamma, \Delta, B \otimes B^\perp} \qquad \dfrac{\overline{\vdash B^\perp, B}}{\vdash B^\perp \parr B, \perp} \qquad \dfrac{\dfrac{\vdash B^\perp, A, B \qquad \overline{\vdash B^\perp, B}}{\vdash B^\perp, B, B \otimes B^\perp, A}}{\vdash B^\perp \parr B, A \parr (B \otimes B^\perp)}}{\vdash \Gamma, \Delta, ¡A}$$

There is a surreptitious twist here: the 'new' internal server protocol is not B, but $B \otimes B^\perp$. This leads to internal back-and-forth communication in the server. Γ is consumed to produce a B. This is 'passed' to each process serving each client. Finally, it is reflected back to the initilization process, and 'finalized' into a Δ. The $\perp$ rule is invertible, so instantiating $\Delta \stackrel{\text{def}}{=} \perp$ in Server gives back the preceding rule. Hence, these two rules are logically equivalent.

3.4 Design Decision #3: Nondeterminism through Permutation

Using list-shaped rules for ¡ forces us to revise the rules for ¿. To define a cut elimination procedure the rules must now match the dual functor K'_A, and hence become

$$\dfrac{}{\vdash ¿A} \qquad\qquad \dfrac{\vdash \Gamma, ¿A \qquad \vdash \Delta, A}{\vdash \Gamma, \Delta, ¿A}$$

The cut elimination procedure for these rules leads to a confluent dynamics. This is unsatisfactory from the perspective of client-server interaction: a proper model requires some nondeterminism in the order in which clients are served. There are many ways to introduce this kind of behaviour. We choose the simplest one: we identify derivations up to permutation of client formation in pools. That is, we quotient them under the least congruence $\equiv$ generated from

$$\dfrac{\dfrac{\vdash \Gamma, ¿A \qquad \vdash \Delta, A}{\vdash \Gamma, \Delta, ¿A} \qquad \vdash \Sigma, A}{\vdash \Gamma, \Delta, \Sigma, ¿A} \equiv \dfrac{\dfrac{\vdash \Gamma, ¿A \qquad \vdash \Sigma, A}{\vdash \Gamma, \Sigma, ¿A} \qquad \vdash \Delta, A}{\vdash \Gamma, \Delta, \Sigma, ¿A}$$

This amounts to quotienting lists up to permutation. Thus, when a client pool interacts with a server, the cut elimination procedure may silently choose to serve any of the constituent clients.

Trees and nondeterminism. The careful reader might notice that the original, tree-like 'distributed contraction' rule $¿c$ inherently supported a certain amount of nondeterminism: if we were to quotient derivations up to permutation of the premises of $¿c$, then the cut elimination procedure would have some choice of whether to serve the left or right subtree first. Switching to list-like functors forbids this move, and seemingly imposes a much stricter discipline.

Nevertheless, the tree structure is awkward and rigid in another way. For example, consider a client pool whose tree structure is informally $[[c_0, c_1], [c_2, c_3]]$. As nondetermistic choices are only made at each node, the clients cannot be served in any order. For example, if c_0 is served first then c_1 must be served next—as it is in the same subtree. From a conventional client-server perspective this is arguably *not* a sufficient amount of nondeterminism. In contrast, our formulation allows full permutations of the client pool.

3.5 Introducing CSLL

Based on the above considerations, we introduce the system CSLL of *Client-Server Linear Logic*.

Following recent presentation of CLL-based systems of session types [Kokke et al. 2019a], CSLL is structured around *hyperenvironments*. Thus the logical system underlying CSLL is not one-sided sequent calculus like CP, but a *hypersequent system* [Avron 1991]. In this kind of presentation process constructors are more finely decoupled. For example, the original CP output/$\otimes$ constructor $x[y].(P \mid Q)$ is a combination of a parallel composition with an output prefix. Hypersequent systems allow us to separately type these two constructs, and bring the language closer to π-calculus.

One-sided sequent systems for CLL—such as Girard's original presentation [1987]—use sequents of the form $\vdash \Gamma$ where Γ is an *environment*, i.e., an unordered list of formulas. We assign distinct *names* to each formula. The environment $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ stands for $A_1 \parr \ldots \parr A_n$. Hence, a comma stands for $\parr$. Environments are identical up to permutation. We write $\cdot$ for the empty one.

A *hyperenvironment* adds another layer: it is an unordered list of environments. We separate environments by vertical lines. If each environment Γ_i stands for the formula A_i, the hyperenvironment $\mathcal{G} = \Gamma_1 \mid \cdots \mid \Gamma_n$ stands for the formula $A_1 \otimes \cdots \otimes A_n$. Hence, $\mid$ stands for $\otimes$. Hyperenvironments are identical up to permutation, and we write $\emptyset$ for the empty one. We also stipulate that variable names be distinct within and across environments.

The syntax and the type system of CSLL are defined in Fig. 1. The types are the formulas of CLL. Note that the choice between curly braces, parantheses and brackets in the syntax of processes is merely typographical, and does not bear formal meaning. However, curly braces are meant to evoke parameters, whereas parentheses and brackets evoke bindings in continuations. A generic judgment of the type system has the shape $P \vdash \mathcal{G}$ where P is a process, and $\mathcal{G}$ is a hyperenvironment.

Most typing rules are identical to HCP, and in the interest of brevity we only discuss the important ones. Hyperenvironment components are introduced by the nullary and binary *hypermix* rules, HMix0 and HMix2. These are 'Mix' rules only in name. HMix2 forms the disjoint parallel composition of two processes: their environments are joined with $\mid$, which stands for $\otimes$.[5] HMix0 is the stopped process; its hyperenvironment is the empty one, which stands for the unit of $\otimes$, namely $\mathbf{1}$.[6]

The Cut and Tensor rules eliminate hyperenvironment components. The premises of Cut ensure that the two variables that are being connected—viz. x and y—are in different 'parallel components' of P. Notice that the external environments of these two components, namely Γ and Δ, are then brought together in the conclusion. A similar pattern permeates the Tensor and M-True rules. It is instructive to follow the derivation of the original CP rules for $\otimes$ and $\mathbf{1}$, which we will silently use:

$$\dfrac{\dfrac{P \vdash \Gamma, y : A \qquad Q \vdash \Delta, x : B}{P \mid Q \vdash \Gamma, y : A \mid \Delta, x : B}}{x[y].(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B} \qquad\qquad \dfrac{\dfrac{}{\mathsf{stop} \vdash \emptyset}}{x[].\,\mathsf{stop} \vdash x : \mathbf{1}}$$

The exponential rules WhyNotW, WhyNotD, WhyNotC and OfCourse are formulated in the style of Kokke et al. [2019a]. In OfCourse we use vector notation ($\vec{-}$) as a shorthand for lists of names and types. Note that—in contrast to all previous systems—we notate P as a parameter rather than as the continuation in the process $!x\{\vec{y};\, P\}$. This because P does not behave like a continuation. For example, it has its own distinct commuting conversion.

The coexponential rules QueW, QueA and Claro follow the patterns described in §§3.1–3.4. The rule QueW (W stands for 'weaken') constructs an empty client pool. The rule QueA (A stands for 'absorb') combines a client and a pool into a slightly larger pool. The interfaces of the client pool and the client are necessarily disjoint, as they are separated by a $\mid$ in the premise. All the processes in the resultant pool race to communicate with a server at the single endpoint x.

Correspondingly, Claro constructs a process that offers a service at the single endpoint y. Its continuation P functions as both the initialization and the finalization of the server, over channels i and f respectively. This rule is similar to the Server rule of §3.3, but in the interest of brevity it

[5] Mix would join them with a comma, which would stand for a $\parr$.
[6] Mix0 would stand for the unit of $\parr$, namely $\bot$.

$$
\begin{array}{lcll}
A, B, \ldots & ::= & \mathbf{1} \mid \bot \mid A \parr B \mid A \otimes B \mid A \oplus B \mid A \,\&\, B \mid ¿A \mid ¡A \mid ?A \mid !A & \\
\Gamma, \Delta, \ldots & ::= & \cdot \mid \Gamma, x : A & \text{(environments)} \\
\mathcal{G}, \mathcal{H}, \ldots & ::= & \emptyset \mid \mathcal{G} \mid \Gamma & \text{(hyperenvironments)} \\
P, Q, \ldots & ::= & \mathsf{stop} & \text{(terminated process)} \\
& \mid & x \leftrightarrow y & \text{(link between } x \text{ and } y\text{)} \\
& \mid & \nu xy.\, P & \text{(connect } x \text{ and } y\text{)} \\
& \mid & P \mid Q & \text{(parallel composition)} \\
& \mid & y.\mathsf{case}\{P; Q\} & \text{(receive choice over } y\text{)} \\
& \mid & y[\mathsf{inl}].\, P \mid y[\mathsf{inr}].\, P & \text{(send choice over } y\text{)} \\
& \mid & y(x).\, P \mid y[x].\, P & \text{(receive/send } x \text{ over } y\text{)} \\
& \mid & y().\, P \mid y[].\, P & \text{(receive/send end-of-session at } y\text{)} \\
& \mid & ¿x[].\, P & \text{(create new client interface } x\text{)} \\
& \mid & ¿x[y].\, P & \text{(send client interface } y \text{ over } x\text{)} \\
& \mid & ¡y\{z', w', y'.\, Q\}(z, w).\, P & \text{(serve over } y\text{)} \\
& \mid & ?x[].\, P \mid ?x[y].\, P \mid ?x[y_0, y_1].\, P & \text{(weakening, dereliction and contraction)} \\
& \mid & !x\{\vec{y};\, P\} & \text{(promotion)}
\end{array}
$$

$$
\textsc{HMix0}\ \frac{}{\mathsf{stop} \vdash \emptyset}
\qquad
\textsc{HMix2}\ \frac{P \vdash \mathcal{G} \qquad Q \vdash \mathcal{H}}{P \mid Q \vdash \mathcal{G} \mid \mathcal{H}}
\qquad
\textsc{Cut}\ \frac{P \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : A^{\bot}}{\nu xy.\, P \vdash \mathcal{G} \mid \Gamma, \Delta}
$$

$$
\textsc{Ax}\ \frac{}{x \leftrightarrow y \vdash x : A^{\bot}, y : A}
\qquad
\textsc{Par}\ \frac{P \vdash \mathcal{G} \mid \Gamma, x : A, y : B}{y(x).\, P \vdash \mathcal{G} \mid \Gamma, y : A \parr B}
\qquad
\textsc{Tensor}\ \frac{P \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : B}{y[x].\, P \vdash \mathcal{G} \mid \Gamma, \Delta, y : A \otimes B}
$$

$$
\textsc{PlusL}\ \frac{P \vdash \mathcal{G} \mid \Gamma, x : A}{x[\mathsf{inl}].\, P \vdash \mathcal{G} \mid \Gamma, x : A \oplus B}
\qquad
\textsc{PlusR}\ \frac{Q \vdash \mathcal{G} \mid \Gamma, y : B}{y[\mathsf{inr}].\, Q \vdash \mathcal{G} \mid \Gamma, y : A \oplus B}
\qquad
\textsc{With}\ \frac{P \vdash \Gamma, x : A \qquad Q \vdash \Gamma, x : B}{x.\mathsf{case}\{P; Q\} \vdash \Gamma, x : A \,\&\, B}
$$

$$
\textsc{M-False}\ \frac{P \vdash \mathcal{G} \mid \Gamma}{x().\, P \vdash \mathcal{G} \mid \Gamma, x : \bot}
\qquad
\textsc{M-True}\ \frac{P \vdash \mathcal{G}}{x[].\, P \vdash \mathcal{G} \mid x : \mathbf{1}}
\qquad
\textsc{WhyNotW}\ \frac{P \vdash \mathcal{G} \mid \Gamma}{?x[].\, P \vdash \mathcal{G} \mid \Gamma, x : ?A}
\qquad
\textsc{WhyNotD}\ \frac{P \vdash \mathcal{G} \mid \Gamma, y : A}{?x[y].\, P \vdash \mathcal{G} \mid \Gamma, x : ?A}
$$

$$
\textsc{WhyNotC}\ \frac{P \vdash \mathcal{G} \mid \Gamma, y_0 : ?A, y_1 : ?A}{?x[y_0, y_1].\, P \vdash \mathcal{G} \mid \Gamma, x : ?A}
\qquad
\textsc{OfCourse}\ \frac{P \vdash \vec{y} : ?\vec{B}, x : A}{!x\{\vec{y};\, P\} \vdash \vec{y} : ?\vec{B}, x : !A}
\qquad
\textsc{QueW}\ \frac{P \vdash \mathcal{G}}{¿x[].\, P \vdash \mathcal{G} \mid x : ¿A}
$$

$$
\textsc{QueA}\ \frac{P \vdash \mathcal{G} \mid \Gamma, x : ¿A \mid \Delta, x' : A}{¿x[x'].\, P \vdash \mathcal{G} \mid \Gamma, \Delta, x : ¿A}
\qquad
\textsc{Claro}\ \frac{P \vdash \mathcal{G} \mid \Gamma, i : B \mid \Delta, f : B^{\bot} \qquad Q \vdash z : B^{\bot}, z' : B, y' : A}{¡y\{z, z', y'.\, Q\}(i, f).\, P \vdash \mathcal{G} \mid \Gamma, \Delta, y : ¡A}
$$

Fig. 1. The syntax and type system of CSLL.

combines the premises $\vdash \Gamma, B$ and $\vdash B^{\perp}, \Delta$ into one process. However, these functionalities continue to be logically disjoint components of P, as their interfaces are separated by a $|$ in the premise. The process Q is a 'worker' process which is spawned every time a client is to be served.

In all process constructs that involve a dot that is not within curly braces, e.g. $y(x).\,P$, we call the part that precedes it the *prefix* of the process ($y(x)$ in this case), and the part that succeeds it the *continuation* (P in this case).

The bound names $\textsc{Bn}(P)$ of a process P are defined as follows:

- x and y are bound in P within $\nu xy.\,P$.
- x is bound in P within $y(x).\,P$ and $y[x].\,P$.
- Within $¡y\{z, z', y'.\,Q\}(i, f).\,P$ we have that i and f are bound in P, while z, z', and y' are bound in Q. Note that y is not bound, but rather 'exported.'
- x is bound in P within $¿y[x].\,P$.
- x_0 and x_1 are bound in P within $?x[x_0, x_1].\,P$.
- x is bound in P within $?x'[x].\,P$.

In all other cases the set of bound names is empty. We define the free names $\textsc{Fn}(P)$ of a process P to be the set of sets corresponding to the names occurring in the typing judgment of P. For example, the hyperenvironment $\mathcal{G} \stackrel{\text{def}}{=} x : A, y : B \mid z : C, w : D$ determines the set of sets $\lfloor \mathcal{G} \rfloor = x, y \mid z, w \stackrel{\text{def}}{=} \{\{x, y\}, \{z, w\}\}$. Kokke et al. [2019a] call this the *name partition* corresponding to a hyperenvironment. Thus, if $P \vdash \mathcal{G}$ we define $\textsc{Fn}(P) \stackrel{\text{def}}{=} \lfloor \mathcal{G} \rfloor$. We will sometimes abusively write $\textsc{Fn}(P)$ to mean the union of the name partition, i.e. the complete set of free names that occur in it. As is usual, processes are identified up to α-equivalence.

We write π_y for an arbitrary prefix communicating on channel y, and $\textsc{Bn}(\pi_y)$ for the variables that it binds in its continuation. For example, π_y could be $y(x)$, and in this case $\textsc{Bn}(\pi_y) = \{x\}$.

Finally, notice that the typing cannot be inferred from the terms alone. For example, in M-False the term $x().\,P$ does not specify in which environment Γ within its hyperenvironment the unit $\perp$ should be introduced. This has an impact on the name partition $\textsc{Fn}(P)$ of a process P.

3.6 Operational Semantics and Metatheory

Definition 3.1. Canonical terms are defined by the following clauses.

- $\pi_x.\,P$ is canonical whenever P is.
- $P \mid Q$ is canonical if both P and Q are canonical.
- stop and $x \leftrightarrow y$ are canonical.
- $y.\mathsf{case}\{P; Q\}$ and $!x\{\vec{y};\ P\}$ are canonical.

In particular, $\nu xy.\,P$ is *not* canonical; it is a cut.

The above notion of canonicity is not definitive. For example, $\pi_x.\,P$ could have been considered canonical regardless of the canonicity of P (similar to weak head normal form for λ-calculus). However, we choose to react P further to make the 'final result' of an interaction visible in later examples. In addition, we could require terms such as P and Q in $y.\mathsf{case}\{P; Q\}$ be canonical for the whole term to be canonical, but we choose not to so as to reduce the number of reaction rules.

We define the notion of *structural equivalence* $P \equiv Q$ to be the least congruence between processes induced by the clauses in Fig. 2. Furthermore, we define the *reaction relation* $P \longrightarrow Q$ between processes to be the least relation induced by the clauses in Fig. 3.

The structural equivalence and the reaction semantics largely mirror the notions of the same name in the π-calculus [Milner 1992, 1999]. Those that differ are justified *via* linear logic. Res-Pre and Pre-Pre can be seen as identifications arising from *proof nets*, in which the corresponding proofs would be graphically identical. Note that the commuting prefixes are requried to not 'cross'

$$
\begin{array}{rlr}
P \mid \mathsf{stop} \equiv P & & \text{(Par-Unit)} \\
P \mid Q \equiv Q \mid P & & \text{(Par-Comm)} \\
P \mid (Q \mid R) \equiv (P \mid Q) \mid R & & \text{(Par-Assoc)} \\
x \leftrightarrow y \equiv y \leftrightarrow x & & \text{(Link-Comm)} \\
\nu xy.\,(P \mid Q) \equiv P \mid \nu xy.\,Q \ (x, y \notin \mathrm{Fn}(P)) & & \text{(Res-Par)} \\
\nu xy.\,\nu zw.\,P \equiv \nu zw.\,\nu xy.\,P & & \text{(Res-Res)} \\
\pi_x.\,(P \mid Q) \equiv P \mid \pi_x.\,Q \ (\mathrm{Bn}(\pi_x) \cap \mathrm{Fn}(P) = \emptyset) & & \text{(Pre-Par)} \\
\nu xy.\,\pi_z.\,P \equiv \pi_z.\,\nu xy.\,P \ (z \neq x, y \text{ and } \pi_z \text{ and } \nu xy. \text{ not cross } \mathrm{Fn}(P)) & & \text{(Res-Pre)} \\
\pi_x.\,\pi_y.\,P \equiv \pi_y.\,\pi_x.\,P \ (x \neq y, y \notin \mathrm{Bn}(\pi_x), x \notin \mathrm{Bn}(\pi_y), \pi_x \text{ and } \pi_y \text{ not cross } \mathrm{Fn}(P)) & & \text{(Pre-Pre)}
\end{array}
$$

extended with

$$¿x[x_0].\,¿x[x_1].\,P \equiv ¿x[x_1].\,¿x[x_0].\,P \qquad \text{(Que-Que)}$$

Fig. 2. The structural equivalence of CSLL processes.

$$
\textsc{ParL}\ \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}
\qquad
\textsc{Res}\ \frac{P \longrightarrow P'}{\nu xy.\,P \longrightarrow \nu xy.\,P'}
\qquad
\textsc{Pre}\ \frac{P \longrightarrow P'}{\pi_y.\,P \longrightarrow \pi_y.\,P'}
\qquad
\textsc{Eq}\ \frac{P \equiv P' \qquad P \longrightarrow Q \qquad Q \equiv Q'}{P' \longrightarrow Q'}
$$

$$
\begin{array}{rlr}
\nu xy.\,(z.\mathsf{case}\{P_0; P_1\} \mid Q) \longrightarrow & z.\mathsf{case}\{\nu xy.\,(P_0 \mid Q); \nu xy.\,(P_1 \mid Q)\} & \text{(With-Comm)} \\
\nu xy.\,(!z\{x\vec{w};\ P\} \mid !y\{\vec{v};\ Q\}) \longrightarrow & !z\{\vec{v}\vec{w};\ \nu xy.\,(P \mid !y\{\vec{v};\ Q\})\} & \text{(OfCourse-Comm)} \\
\nu xy.\,(z \leftrightarrow x \mid Q) \longrightarrow & Q[z/y] & \text{(Link)} \\
\nu xy.\,(x[].\,P \mid y().\,Q) \longrightarrow & P \mid Q & \text{(One-Bot)} \\
\nu xy.\,(x[z].\,P \mid y(w).\,Q) \longrightarrow & \nu xy.\,\nu zw.\,(P \mid Q) & \text{(Tensor-Par)} \\
\nu xy.\,(x[\mathsf{inl}].\,P \mid y.\mathsf{case}\{Q_0; Q_1\}) \longrightarrow & \nu xy.\,(P \mid Q_0) & \text{(PlusL-With)} \\
\nu xy.\,(x[\mathsf{inr}].\,P \mid y.\mathsf{case}\{Q_0; Q_1\}) \longrightarrow & \nu xy.\,(P \mid Q_1) & \text{(PlusR-With)} \\
\nu xy.\,(¿x[].\,C \mid ¡y\{z, z', y'.\,Q\}(i, f).\,P) \longrightarrow & C \mid \nu if.\,P & \text{(Claro-QueW)} \\
\nu xy.\,(¿x[x'].\,C \mid ¡y\{z, z', y'.\,Q\}(i, f).\,P) \longrightarrow & \nu xy.\,\nu x'y'.\,(C \mid ¡y\{z, z', y'.\,Q\}(z', f).\,(\nu iz.\,(P \mid Q))) & \\
 & & \text{(Claro-QueA)} \\
\nu xy.\,(?x[].\,P \mid !y\{\vec{z};\ Q\}) \longrightarrow & ?\vec{z}[].\,P & \text{(ExpW)} \\
\nu xy.\,(?x[x'].\,P \mid !y\{\vec{z};\ Q\}) \longrightarrow & \nu x'y.\,(P \mid Q) & \text{(ExpD)} \\
\nu xy.\,(?x[x_0, x_1].\,P \mid !y\{\vec{z};\ Q\}) \longrightarrow & ?\vec{z}[\vec{z}_0, \vec{z}_1].\,\nu x_0 y_0.\,\nu x_1 y_1.\,(P \mid R) & \\
 & \text{where } R \stackrel{\text{def}}{=} !y_1\{\vec{z}_1;\ Q[\vec{z}_1 y_1/\vec{z}y]\} \mid !y_0\{\vec{z}_0;\ Q[\vec{z}_0 y_0/\vec{z}y]\} & \text{(ExpC)}
\end{array}
$$

Fig. 3. The operational semantics of CSLL processes.

the name partition in order to preserve typing. As a counterexample, if $P \vdash x : A, y : B \mid z : C, w : D$, then $x(w).\,y[z].\,P \vdash x : A ⅋ D, y : B \otimes C$ while $y[z].\,x(w).\,P$ is ill-typed. To avoid this, we say that $x(w).$ and $y[z].$ cross the name partition $\mathrm{Fn}(P) = x, y \mid z, w$, and hence that this commutation is forbidden. Formally, 'crossing' is defined as follows.

Definition 3.2. We first define two sets of names X_{π_x} and Y_{π_y} indexed by prefixes. Note they are defined for only some prefixes.

$$X_{x(x').} \stackrel{\text{def}}{=} \{x, x'\} \qquad X_{?x[x_0,x_1].} \stackrel{\text{def}}{=} \{x_0, x_1\}$$

$$Y_{y[y'].} \stackrel{\text{def}}{=} \{y, y'\} \qquad Y_{¿y[y'].} \stackrel{\text{def}}{=} \{y, y'\}$$

$$Y_{¡y\{z',w',y'.Q\}(z,w).} \stackrel{\text{def}}{=} \{z, w\}$$

Now, π_x and π_y cross the name partition $\lfloor \mathcal{G} \rfloor$ just if any of the following cases apply.

- In the binary case, we require the following: X_{π_x} and Y_{π_y} is defined for π_x and π_y respectively; write $X_{\pi_x} = \{x_0, x_1\}$ and $Y_{\pi_y} = \{y_0, y_1\}$; there are $\Gamma, \Delta \in \lfloor \mathcal{G} \rfloor$ such that $x_0, y_0 \in \Gamma$ and $x_1, y_1 \in \Delta$.
- In the nullary case, we require all the following to hold:
  - π_x is $x().$ or $?x[].$
  - π_y is $y[].$ or $¿x[].$
  - $\lfloor \mathcal{G} \rfloor$ is $\emptyset$

Moreover, π_x and $\nu y_0 y_1.$ cross the name partition $\lfloor \mathcal{G} \rfloor$ if the following holds: X_{π_x} is defined for π_x; write $X = \{x_0, x_1\}$ and there is $\Gamma, \Delta \in \lfloor \mathcal{G} \rfloor$ such that $x_0, y_0 \in \Gamma$ and $x_1, y_1 \in \Delta$.

The structural equivalence Que-Que allows us to commute the position of two clients in the pool, thereby imitating racing—as discussed in §3.4. In order fully exploit the nondeterminism induced by Que-Que the other structural equivalences are necessary. For example, the two clients in $¿x[x_0].\, y(y').\, ¿x[x_1].\, P$ cannot be permuted without using Pre-Pre first. Indeed, this is the major motivation for Pre-Pre, as the latter is not needed for our metatheoretic results. Note that Que-Que is the one and only source of nondeterminism in the system.

Some commuting conversions appear as structural equivalences, and some as reaction rules. OfCourse-Comm and With-Comm are commuting conversions for OfCourse and With respectively. Pre-Par with Res-Pre combine into a kind of commuting conversion for prefixes. We take the former as reaction rules, and the latter as structural equivalences. This choice makes structural equivalence preserve canonicity. For example, in With-Comm the LHS is not canonical, but the RHS is.

The overwhelming majority of these commuting conversions is used in previous works on the relationship between linear logic and π-calculus to obtain cut elimination [Wadler 2014, §3.6] [Bellin and Scott 1994, §3]. Perhaps the only exception is Pre-Pre, which allows us to swap any two noninterfering prefixes. It can be justified computationally as an observational equivalence arising from the semantics of Atkey [2017, §5]. Finally, Kokke et al. [2019a] view it as a session-theoretic version of *delayed actions* [Merro and Sangiorgi 2004].

Pre corresponds to eliminating non-top-level cuts in Linear Logic; it is not standard in either π-calculus or CP. Nevertheless, we choose to include it in order to strengthen our notion of canonical form, which in turn elucidates the examples in §4. In contrast, the reaction rules for the exponentials are standard; see Kokke et al. [2019a].

Finally, we have a number of novel reaction rules for coexponentials. The rule Claro-QueW corresponds to serving an empty client pool. In this case we simply connect the initialization and finalization channels of P. Likewise, the rule Claro-QueA is the reaction caused by a nonempty pool of clients. The pool offers a fresh channel x' on which the new client expects to be served. The server then spawns a worker process Q, and the channel y' on which it will serve the new client which is connected to x', as expected. The initialization channel i of the server continuation is connected to the z channel, on which the worker process expects to receive the 'current state' of the server. Once Q serves the client, it will send the 'next state' of the server on z'. Thus, we re-instantiate the server with z' as the new initialization channel. Note that the 'server state' we discuss here does not conform to the usual intuition of an immutable value; it could be a session type itself, as demonstrated by the example in §5.5.

We have the following metatheoretic results.

Lemma 3.3. *If $P \equiv Q$, then $P \vdash \mathcal{G}$ if and only if $Q \vdash \mathcal{G}$.*

Theorem 3.4 (Preservation). *If $P \vdash \mathcal{G}$ and $P \longrightarrow Q$, then $Q \vdash \mathcal{G}$.*

Theorem 3.5 (Progress). *If $R \vdash \mathcal{G}$ then either R is canonical, or there exists R' such that $R \longrightarrow R'$.*

4 AN EXAMPLE: COMPARE-AND-SET

We now wish to demonstrate the client-server features of CSLL. To do so we produce an implementation of the quintessential example of a synchronization primitive, the *Compare-and-Set operation* (CAS) [Herlihy and Shavit 2012, §5.8]. Higher-level examples are given in §5.

A register that supports compare-and-set comes with an operation $\textsc{Cas}(e, d)$ which takes two values: the *expected* value e, and the *desirable* value d. The function compares the expected value e with the register. If the two differ, the value of the register remains put, and $\textsc{Cas}(e, d)$ returns false. But if they are found equal, the register is updated with the desirable value d, and $\textsc{Cas}(e, d)$ returns true. When multiple clients are trying to perform CAS operations on the same register they must be performed *atomically*. The CAS operation is very powerful: an asynchronous machine that supports it can implement all concurrent objects in a wait-free manner.

We follow previous work [Abramsky 1993a; Atkey et al. 2016; Girard 1987; Kokke et al. 2019a] and define the type of Boolean sessions to be $\mathbf{2} \stackrel{\text{def}}{=} \mathbf{1} \oplus \mathbf{1}$. We have the following derivable constants:

$$\mathsf{tt}_z \stackrel{\text{def}}{=} z[\mathsf{inl}].\, z[].\, \mathsf{stop} \vdash z : \mathbf{2} \qquad\qquad \mathsf{ff}_z \stackrel{\text{def}}{=} z[\mathsf{inr}].\, z[].\, \mathsf{stop} \vdash z : \mathbf{2}$$

Moreover, we obtain the following derivable 'elimination' rule (we write derivable rules in blue):

$$\dfrac{\dfrac{P \vdash \Gamma}{z().\, P \vdash z : \bot, \Gamma} \qquad \dfrac{Q \vdash \Gamma}{z().\, Q \vdash z : \bot, \Gamma}}{\mathsf{if}(z;\ P;\ Q) \stackrel{\text{def}}{=} z.\mathsf{case}\{z().\, P; z().\, Q\} \vdash z : \mathbf{2}^{\bot}, \Gamma}$$

Hence, we can eliminate a Boolean channel in any environment Γ. The induced reactions are

$$\nu xy.\, (\mathsf{tt}_x \mid \mathsf{if}(y;\ P;\ Q)) \longrightarrow^{*} \mathsf{stop} \mid P \equiv P \qquad \nu xy.\, (\mathsf{ff}_x \mid \mathsf{if}(y;\ P;\ Q)) \longrightarrow^{*} \mathsf{stop} \mid Q \equiv P$$

We can now implement a register with a CAS operation. To begin, each client communicates with the register along a channel of type

$$A \stackrel{\text{def}}{=} \mathbf{2} \otimes \mathbf{2} \otimes \mathbf{2}^{\bot} \invamp \mathbf{1}$$

Thus, a client outputs three channels. On the first two it shall send the expected and desirable values. On the third it will input a boolean, namely the success flag of the CAS operation. Following that, it will accept an end-of-session signal. Curiously, this last step is necessary for our implementation to type-check.

As a minimal example we will construct a pool of two racing clients, one performing $\textsc{Cas}(\mathsf{ff}, \mathsf{tt})$, and the other one $\textsc{Cas}(\mathsf{tt}, \mathsf{ff})$. Initially x_1 is ahead in the client pool.

$$C_0 \stackrel{\text{def}}{=} x_0[x_e].\, x_0[x_d].\, (\mathsf{ff}_{x_e} \mid \mathsf{tt}_{x_d} \mid x_0 \leftrightarrow r_0) \vdash x_0 : \mathbf{2} \otimes \mathbf{2} \otimes \mathbf{2}^{\bot} \invamp \mathbf{1}, r_0 : \mathbf{2} \otimes \bot$$

$$C_1 \stackrel{\text{def}}{=} x_1[x_e].\, x_1[x_d].\, (\mathsf{tt}_{x_e} \mid \mathsf{ff}_{x_d} \mid x_1 \leftrightarrow r_1) \vdash x_1 : \mathbf{2} \otimes \mathbf{2} \otimes \mathbf{2}^{\bot} \invamp \mathbf{1}, r_1 : \mathbf{2} \otimes \bot$$

$$\mathsf{clients} \stackrel{\text{def}}{=} \text{¿}x[x_1].\, \text{¿}x[x_0].\, \text{¿}x[].\, (C_0 \mid C_1) \vdash x : \text{¿}\left(\mathbf{2} \otimes \mathbf{2} \otimes \mathbf{2}^{\bot} \invamp \mathbf{1}\right), r_0 : \mathbf{2} \otimes \bot, r_1 : \mathbf{2} \otimes \bot$$

Note that each client forwards the result it receives to an individual channel r_i. By the QueA rule these two channels are preserved in the final interface of the pool.

Next we define the CAS register process, for which we use the ¡ connective. This requires two components: the initialization and finalization process P, and the worker process Q that serves one client. To begin, we pick the internal server state to be $B \stackrel{\text{def}}{=} \mathbf{2}$. We initialize the register to false, and forward the final state of the register to u.

$$P \stackrel{\text{def}}{=} (\mathsf{ff}_i \mid f \leftrightarrow u) \vdash i : \mathbf{2} \mid f : \mathbf{2}^{\perp}, u : \mathbf{2}$$

Finally, we define Q. We begin by receiving the input and output channels from a client, and do a case analysis on the current state of the register:

$$Q \stackrel{\text{def}}{=} y'(y_e).\, y'(y_d).\, \mathsf{if}(z;\, R_1;\, R_0) \vdash z : \mathbf{2}^{\perp}, y' : \mathbf{2}^{\perp} \parr \mathbf{2}^{\perp} \parr \mathbf{2} \otimes \perp, z' : \mathbf{2}$$

We have carefully named the channels so that $y_e : \mathbf{2}^{\perp}$ and $y_d : \mathbf{2}^{\perp}$ carry the expected and desirable values. z' and w' carry the internal register, before and after the operation. The continuations R_0 and R_1 do a case analysis on the expected and desired value:

$$R_1 \stackrel{\text{def}}{=} \mathsf{if}(y_e;\, \mathsf{if}(y_d;\, S_{111};\, S_{110});\, \mathsf{if}(y_d;\, S_{101};\, S_{100})) \vdash y_e : \mathbf{2}^{\perp}, y_d : \mathbf{2}^{\perp}, y' : \mathbf{2} \otimes \perp, z' : \mathbf{2}$$

$$R_0 \stackrel{\text{def}}{=} \mathsf{if}(y_e;\, \mathsf{if}(y_d;\, S_{011};\, S_{010});\, \mathsf{if}(y_d;\, S_{001};\, S_{000})) \vdash y_e : \mathbf{2}^{\perp}, y_d : \mathbf{2}^{\perp}, y' : \mathbf{2} \otimes \perp, z' : \mathbf{2}$$

Two further case analyses lead to an exhaustive eight cases, each of which is handled by a separate process S_{ijk}. We only give S_{110} here, the rest being analogous:

$$S_{110} \stackrel{\text{def}}{=} y'[y_r].\, (\mathsf{tt}_{y_r} \mid y'().\, \mathsf{ff}_{z'}) \vdash y' : \mathbf{2} \otimes \perp, z' : \mathbf{2}$$

In this case, the expected value (true) matches the register state (true), so the process outputs true to the result channel y_r (the CAS operation succeeds), and the register is set to the desired value (false). We must not forget to receive an end-of-session signal on y, as required by the session type. We let $\mathsf{server} \stackrel{\text{def}}{=} ¡y\{z, z', y'.\, Q\}(i, f).\, P \vdash y : ¡(\mathbf{2}^{\perp} \parr \mathbf{2}^{\perp} \parr \mathbf{2} \otimes \perp), u : \mathbf{2}$, and cut:

$$\begin{aligned}
& \nu xy.\, (\mathsf{clients} \mid \mathsf{server}) \\
&= \nu xy.\, (¿x[x_1].\, ¿x[x_0].\, ¿x[].\, (C_0 \mid C_1) \mid \mathsf{server}) \\
&\equiv \nu xy.\, (¿x[x_0].\, ¿x[x_1].\, ¿x[].\, (C_0 \mid C_1) \mid \mathsf{server}) && (x_0 \text{ preempts } x_1 \text{ using Que-Que}) \\
&\longrightarrow \nu xy.\, \nu x_0 y'.\, (C_0 \mid ¿x[x_1].\, ¿x[].\, C_1 \mid ¡y\{z, z', y'.\, Q\}(z', f).\, (\nu iz.\, (P \mid Q))) && (C_0 \text{ is accepted}) \\
&\longrightarrow^* r_0[y_r].\, (\mathsf{tt}_{y_r} \mid r_0().\, \nu xy.\, (¿x[x_1].\, ¿x[].\, C_1 \mid ¡y\{z, z', y'.\, Q\}(z'', f).\, P')) && (C_0 \text{ performs CAS}) \\
&\longrightarrow r_0[y_r].\, (\mathsf{tt}_{y_r} \mid r_0().\, \nu xy.\, \nu x_1 y'.\, (C_1 \mid ¿x[].\, \mathsf{stop} \mid ¡y\{z, z', y'.\, Q\}(z', f).\, (\nu z'' z.\, (P' \mid Q)))) \\
& && (C_1 \text{ is accepted}) \\
&\longrightarrow^* r_0[y_r].\, (\mathsf{tt}_{y_r} \mid r_0().\, r_1[y_r].\, (\mathsf{tt}_{y_r} \mid r_1().\, \nu xy.\, (¿x[].\, \mathsf{stop} \mid ¡y\{z, z', y'.\, Q\}(z''', f).\, P''))) \\
& && (C_1 \text{ performs CAS}) \\
&\longrightarrow r_0[y_r].\, (\mathsf{tt}_{y_r} \mid r_0().\, r_1[y_r].\, (\mathsf{tt}_{y_r} \mid r_1().\, (\mathsf{stop} \mid \nu z''' f.\, P''))) && (\mathsf{server} \text{ starts to finalize}) \\
&\longrightarrow^* r_0[y_r].\, (\mathsf{tt}_{y_r} \mid r_0().\, r_1[y_r].\, (\mathsf{tt}_{y_r} \mid r_1().\, \mathsf{ff}_u)) \vdash r_0 : \mathbf{2} \otimes \perp, r_1 : \mathbf{2} \otimes \perp, u : \mathbf{2} && (\mathsf{server} \text{ finalizes})
\end{aligned}$$

where $P' = \mathsf{tt}_{z''} \mid f \leftrightarrow u$ and $P'' = \mathsf{ff}_{z'''} \mid f \leftrightarrow u$. This corresponds to the scenario where C_0 wins the first race, and hence the CAS operation of both clients suceeds. There is another reaction sequence: if C_1 wins the first race, we end up with $r_1[y_r].\, (\mathsf{ff}_{y_r} \mid r_1().\, r_0[y_r].\, (\mathsf{tt}_{y_r} \mid r_0().\, \mathsf{tt}_u))$.

The coexponentials play a central rôle here: ¡ is used to represent the fact that this register provides a server session at a unique end point, and ¿ is used to collect requests for a CAS operation to this single end point. We see that every feature of client-server interaction, as described in points (i)–(iv) of §1.1, is modelled.

The fact we are able to implement a synchronization primitive like CAS shows that the client-server rules also provide an additional safeguard, namely that *server acceptance is atomic*. While the actual CAS is *not* an atomic operation—as many things are happening in parallel—the causal flow of information ensures that the state implicitly remains atomic.

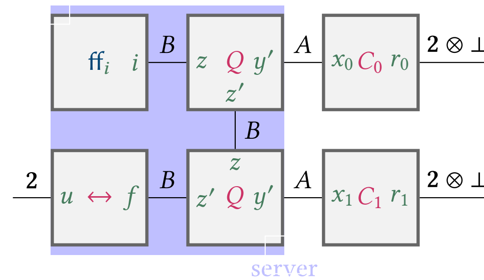

Fig. 4. Topology of Compare-and-Set protocol, after two server acceptances. Boxes represent processes. Cuts are represented by edges connecting two channels. The dual of each session type is omitted for simplicity.

To illustrate the type of atomicity we have, consider an alternative reaction sequence where the two clients are immediately accepted before any other reaction. Fig. 4 shows the process topology of the scenario where C_0 is accepted immediately before C_1. Each client is connected to the one of the two worker processes Q with client protocol A, and the worker processes are connected to each other and P with internal server protocol B. Which specific worker process a client connects to is determined by the client's position in the queue, before the coexponential reaction Claro-QueA takes place. The clients' positions in the layout also determine the final result of the reaction up to structural equivalence, even before the computation of the output takes place.

5 A SESSION-TYPED LANGUAGE FOR CLIENT-SERVER PROGRAMMING

As the example of the previous section shows, CSLL is a particularly low-level language. This is a feature of essentially all variants of linear logic as used for session typing, including Kokke et al.'s HCP [2019a, Example 2.1], and Wadler's CP [Atkey 2017, §2.1] [Atkey et al. 2016, §3.1]. Consequently, the need for higher-level notation to help us write richer examples arises. These in turn will help us illustrate the degree of channel sharing allowed by CSLL. We follow the lead of Wadler [2014, §4] and introduce a higher-level, session-typed functional language, which we call CSGV.

CSGV is a linear λ-calculus augmented with session types and communication primitives. It is based on the influential work of Gay and Vasconcelos [2010]. Over the past decade many variations of this language have been proposed; see e.g. Lindley and Morris [2015, 2016, 2017] and Fowler et al. [2019]. CSGV extends Wadler's version with primitives for client-server interaction. Like the approach in *loc. cit.* we do not directly endow CSGV with a semantics. Instead, we formulate a type-preserving translation into CSLL, which indirectly provides an execution mechanism. Naturally, the client-server primitives translate to the coexponential rules of CSLL.

5.1 Source Language and the Translation

Types. The types of CSGV consist of standard functional types and session types. While the former are used to classify values, the latter are used to describe the behaviour of channels. Compared to Wadler [2014] we have added sum types, and session types for client-server shared channels.

$$
\begin{array}{lll}
T, \ldots & ::= & T \multimap T \mid T \to T \mid T + T \mid T \otimes T \mid \mathsf{Unit} \mid T_S \\
T_S, \ldots & ::= & !T.T_S \qquad \text{(output value of type } T\text{, then behave as } T_S) \\
& & \mid ?T.T_S \qquad \text{(input value of type } T\text{, then behave as } T_S) \\
& & \mid T_S \oplus T_S \qquad \text{(select from options)}
\end{array}
$$

$$
\begin{aligned}
&\mid T_S \mathbin{\&} T_S && \text{(offer choice)}\\
&\mid \mathsf{end}_? \mid \mathsf{end}_! && \text{(end-of-session)}\\
&\mid ¿T_S && \text{(request } T_S \text{ session)}\\
&\mid ¡T_S && \text{(serve } T_S \text{ session)}
\end{aligned}
$$

Both the functional types and the session types of CSGV are translated to the linear types of CSLL. The functional part closely follows Wadler in using the ‘call-by-value’ embedding of intuitionistic logic into linear logic [Benton and Wadler 1996; Maraist et al. 1999, 1995]. The session types are translated as follows:

$$
\begin{array}{lll}
[\![!T.T_S]\!] \stackrel{\text{def}}{=} [\![T]\!]^\perp \mathbin{⅋} [\![T_S]\!] & [\![T_S \mathbin{\&} U_L]\!] \stackrel{\text{def}}{=} [\![T_S]\!] \oplus [\![U_L]\!] & [\![\mathsf{end}_!]\!] \stackrel{\text{def}}{=} \perp \\
[\![?T.T_S]\!] \stackrel{\text{def}}{=} [\![T]\!] \otimes [\![T_S]\!] & [\![T_S \oplus U_L]\!] \stackrel{\text{def}}{=} [\![T_S]\!] \mathbin{\&} [\![U_L]\!] & [\![\mathsf{end}_?]\!] \stackrel{\text{def}}{=} \mathbf{1} \\
[\![¿T_S]\!] \stackrel{\text{def}}{=} ¡[\![T_S]\!] & [\![¡T_S]\!] \stackrel{\text{def}}{=} ¿[\![T_S]\!] &
\end{array}
$$

As noted by Wadler [2014, §4.1], the connectives translate to the dual of what one might expect. The reason is that channels are used in the opposite way. Consider the session type $!T.S$: sending a value in CSGV is translated as inputting a channel on which you can send it in CSLL. Similarly, $¡S$ does not represent a channel that the server provides, but rather a channel that the server consumes. It is therefore a channel that the client pool provides, and hence it is translated to a client in CSLL.

Duality. We define duality on session types in the standard way; it is obviously an involution.

$$
\begin{array}{lll}
\overline{!T.T_S} \stackrel{\text{def}}{=} ?T.\overline{T_S} & \overline{!T.T_S} \stackrel{\text{def}}{=} ?T.\overline{T_S} & \overline{T_S \oplus U_L} \stackrel{\text{def}}{=} \overline{T_S} \mathbin{\&} \overline{U_L} \\
\overline{T_S \mathbin{\&} U_L} \stackrel{\text{def}}{=} \overline{T_S} \oplus \overline{U_L} & \overline{¿T_S} \stackrel{\text{def}}{=} ¡\overline{T_S} & \overline{¡T_S} \stackrel{\text{def}}{=} ¿\overline{T_S}
\end{array}
$$

The translation is a homomorphism of involutions:

Lemma 5.1. $[\![\overline{T_S}]\!] = [\![T_S]\!]^\perp$.

Thus, connecting channels in CSGV will be translated to cuts in linear logic.

Definition 5.2. The set of *unlimited types* is defined inductively as follows.

- unit and $T \longrightarrow U$ are unlimited.
- $T + U$ and $T \otimes U$ are unlimited whenever T and U are.

All other types are *linear*.

Values of unlimited types can be discarded and duplicated, because they are translated to CSLL types that admit weakening and contraction. Categorical considerations [Melliès 2009, §6.5] lead us to consider $T \otimes U$ unlimited whenever T and U are, which is finer-grained than *loc. cit.*

Terms. CSGV is a linear λ-calculus, extended with constructs for sending and receiving messages.

$$
\begin{aligned}
L, M, N ::= \quad & x \mid \star \mid \lambda x.\, N \mid MN \mid (M, N) \mid \mathsf{let}\ (x, y) \ = \ M \ \mathsf{in}\ N \\
& \mid \mathsf{inl}\ M \mid \mathsf{inr}\ M \mid \mathsf{match}\ L\ \mathsf{with}\ x.\{M, N\} && \text{(functional fragment)}\\
& \mid \mathsf{send}\ M\ N \mid \mathsf{recv}\ M && \text{(send and receive)}\\
& \mid \mathsf{select}_L\ M \mid \mathsf{select}_R\ M \mid \mathsf{case}\ L\ \mathsf{of}\ x.\{M, N\} && \text{(select options)}\\
& \mid \mathsf{terminate}\ M && \text{(terminate } M\text{)}\\
& \mid \mathsf{connect}(x.\, M; y.\, N) && \text{(connect } x \text{ of } M \text{ to } y \text{ of } N\text{)}\\
& \mid \mathsf{eof}_x && \text{(end client pool)}\\
& \mid \mathsf{fork}_x\ x'.\, M && \text{(extract client interface)}
\end{aligned}
$$

$$\left[\!\!\left[\frac{\Gamma \vdash M : ?T.T_S}{\Gamma \vdash \mathsf{recv}\ M : T \otimes T_S}\ \textsc{Recv} \right]\!\!\right]_z \stackrel{\text{def}}{=} [\![M]\!]_z \vdash [\![\Gamma]\!]^\perp, z : [\![T]\!] \otimes [\![T_S]\!]$$

$$\left[\!\!\left[\frac{\Gamma \vdash M : T \qquad \Delta \vdash N : !T.T_S}{\Gamma, \Delta \vdash \mathsf{send}\ M\ N : T_S}\ \textsc{Send} \right]\!\!\right]_z \stackrel{\text{def}}{=}$$

$$\frac{\dfrac{[\![M]\!]_y \vdash [\![\Gamma]\!]^\perp, y : [\![T]\!] \qquad x' \leftrightarrow z \vdash x' : [\![T_S]\!]^\perp, z : [\![T_S]\!]}{x'[y].\,([\![M]\!]_y \mid x' \leftrightarrow z) \vdash [\![\Gamma]\!]^\perp, x' : [\![T]\!] \otimes [\![T_S]\!]^\perp, z : [\![T_S]\!]}\ \otimes \qquad [\![N]\!]_x \vdash [\![\Delta]\!]^\perp, x : [\![T]\!]^\perp \parr [\![T_S]\!]}{\nu x x'.\,(x'[y].\,([\![M]\!]_y \mid x' \leftrightarrow z) \mid [\![N]\!]_x) \vdash [\![\Gamma]\!]^\perp, [\![\Delta]\!]^\perp, z : [\![T_S]\!]}$$

$$\left[\!\!\left[\frac{\Gamma, x : T_S \vdash M : \mathsf{end}_! \qquad \Delta, y : \overline{T_S} \vdash N : T}{\Gamma, \Delta \vdash \mathsf{connect}(x.\,M; y.\,N) : T}\ \textsc{Conn} \right]\!\!\right]_z \stackrel{\text{def}}{=}$$

$$\frac{\dfrac{[\![M]\!]_y \vdash [\![\Gamma]\!]^\perp, x : [\![T_S]\!]^\perp, y : \perp \qquad z[].\,\mathsf{stop} \vdash z : \mathbf{1}}{\nu y z.\,([\![M]\!]_y \mid z[].\,\mathsf{stop}) \vdash [\![\Gamma]\!]^\perp, x : [\![T_S]\!]^\perp}\ \textsc{Cut} \qquad [\![N]\!]_z \vdash [\![\Delta]\!]^\perp, y : [\![T_S]\!], z : [\![T]\!]}{\nu x y.\,(\nu y z.\,([\![M]\!]_y \mid z[].\,\mathsf{stop}) \mid [\![N]\!]_z) \vdash [\![\Gamma]\!]^\perp, [\![\Delta]\!]^\perp, z : [\![T]\!]}\ \textsc{Cut}$$

Fig. 5. CSGV Typing Rules and Translation to CSLL: linear session part

$$\mid \mathsf{serve}\ y\{L, z.\,M, f.\,N\} \qquad \text{(server construction)}$$

Typing rules. The environments of CSGV are given by $\Gamma, \cdots ::= \bullet \mid \Gamma, x : T$. The translation of types is extended to environments pointwise.

Selected typing rules of CSGV are given in Figs. 5 and 6. Most rules follow Wadler [2014, §4.1] to the letter, and are therefore omitted. In the interest of economy we also give the translation to CSLL at the same time. The translation is defined by induction on the typing derivations of CSGV. As the purpose of a CSGV program is the computation of a value of a distinguished type, the translation must privilege a single name over which this value will be returned. Thus, given a choice of name z and a typing derivation $\Gamma \vdash M : T$, we write $[\![\Gamma \vdash M : T]\!]_z$ for its translation into CSLL. Somewhat abusively we will sometimes also write $[\![M]\!]_z \vdash [\![\Gamma]\!]^\perp, z : [\![T]\!]$ for the translated term. This slight abuse of notation also reveals the intended typing.

The novelty here is in the CSGV rules for client-server interaction, and their translation into CSLL. A name of shared client type $¿T_S$ can be seen as a form of 'capability' for talking to the server. ReqW discards this capability, signalling the end of the client pool. ReqA uses it to spawn a fresh channel x' on which a client M will talk to a server, and returns the capability back to the caller. The client M itself has type $\mathsf{end}_!$: it does not return valuable information, but uses values and channels found in Γ.

Dually, $¡T_S$ is the type of a server channel. Serv constructs a server from three components. L computes the initial state of the server. Given the current state in z, and a client channel y, M serves the client listening on y, and then returns the next state of the server. N finalizes the server. Note that the so-called server 'state' here could well be a channel itself, enabling bidirectional interleaving communication—a design we will explore in §5.5.

The Serv typing rule is quite restrictive, in that it does not allow anything from the environments Δ and Σ to be used in the term M which computes the next state of the server. Fortunately, the

$$\left[\!\!\left[\dfrac{\textsc{ReqW}}{x : ¿T_S \vdash \mathsf{eof}_x : \mathsf{end}_!}\right]\!\!\right]_z \overset{\text{def}}{=} \dfrac{\dfrac{\mathsf{stop} \vdash \emptyset}{¿x[].\,\mathsf{stop} \vdash x : ¿[\![T_S]\!]^\perp}}{z().\,¿x[].\,\mathsf{stop} \vdash x : ¿[\![T_S]\!]^\perp, z : \perp}$$

$$\left[\!\!\left[\dfrac{\textsc{ReqA}\quad \Gamma, x' : T_S \vdash M : \mathsf{end}_!}{\Gamma, x : ¿T_S \vdash \mathsf{fork}_x\ x'.\,M : ¿T_S}\right]\!\!\right]_z \overset{\text{def}}{=}$$

$$\dfrac{\dfrac{[\![M]\!]_u \vdash [\![\Gamma]\!]^\perp, x' : [\![T_S]\!]^\perp, u : \perp \qquad v[].\,\mathsf{stop} \vdash v : \mathbf{1}}{\nu uv.\,([\![M]\!]_z \mid v[].\,\mathsf{stop}) \vdash [\![\Gamma]\!]^\perp, x' : [\![T_S]\!]^\perp} \qquad x \leftrightarrow z \vdash x : ¿[\![T_S]\!]^\perp, z : ¡[\![T_S]\!]}{¿x[x'].\,(\nu uv.\,(z[].\,\mathsf{stop} \mid [\![M]\!]_u) \mid x \leftrightarrow z) \vdash [\![\Gamma]\!]^\perp, x : ¿[\![T_S]\!]^\perp, z : ¡[\![T_S]\!]}\ \text{HMix2+QueA}$$

$$\left[\!\!\left[\dfrac{\textsc{Serv}\quad \Delta \vdash L : T \qquad z : T, y : T_S \vdash M : T \qquad \Sigma, f : T \vdash N : U}{\Delta, \Sigma, y : ¡T_S \vdash \mathsf{serve}\ y\{L, z.\,M, f.\,N\} : U}\right]\!\!\right]_u \overset{\text{def}}{=}$$

$$\dfrac{\dfrac{[\![L]\!]_i \vdash [\![\Delta]\!]^\perp, i : [\![T]\!] \qquad [\![N]\!]_u \vdash [\![\Sigma]\!]^\perp, f : [\![T]\!]^\perp, u : [\![U]\!]}{[\![L]\!]_i \mid [\![N]\!]_u \vdash [\![\Delta]\!]^\perp, i : [\![T]\!] \mid [\![\Sigma]\!]^\perp, f : [\![T]\!]^\perp, u : [\![U]\!]} \qquad [\![M]\!]_{z'} \vdash z : [\![T]\!]^\perp, y : [\![T_S]\!]^\perp, z' : [\![T]\!]}{¡y\{z, z', y.\,[\![M]\!]_{z'}\}(i, f).\,([\![L]\!]_i \mid [\![N]\!]_u) \vdash [\![\Delta]\!]^\perp, y : ¡[\![T_S]\!]^\perp, [\![\Sigma]\!]^\perp, u : [\![U]\!]}\ \textsc{Claro}$$

Fig. 6. CSGV Typing Rules and Translation to CSLL: shared session part

following derivable rule allows us to weave some non-linear values of types $\vec{V}$ in the server.

$$\dfrac{\Delta \vdash L : T \qquad \vec{v} : \vec{V}, z : T, y : T_S \vdash M : T \qquad \Sigma, f : T \vdash N : U \qquad \vec{V}\ \text{unlimited}}{\vec{v} : \vec{V}, \Delta, \Sigma, y : ¡T_S \vdash \underbrace{\mathsf{serve}\ y\{(\vec{v}, L), z'.\,\mathsf{let}\ (\vec{v}, z) = z'\ \mathsf{in}\ (\vec{v}, M), f'.\,\mathsf{let}\ (\vec{v}, f) = f'\ \mathsf{in}\ N\}}_{\mathsf{serve'}\ y\{L, z.\,M, f.\,N\}} : U}$$

We will make crucial use of this derivable rule in a couple of our examples. We also also adopt the common shorthands $\mathsf{let}\ x = M\ \mathsf{in}\ N \overset{\text{def}}{=} (\lambda x.\,N)M$ and $\mathsf{let}\ \_ = M\ \mathsf{in}\ N \overset{\text{def}}{=} (\lambda z.\,N)M$ for fresh $z : \star$.

5.2 Functional Data Structure Server

Our primitives can be used to protect a shared functional data structure. Without loss of generality, we consider a server whose state is a purely functional queue T with operations

$$\mathsf{enq} : T \otimes A \to T \qquad \mathsf{deq} : T \to T \otimes (\mathsf{Unit} + A) \qquad \mathsf{empty} : T$$

In particular, deq could return Unit if the queue is empty. The server will talk to a client via a channel of type $T_S \overset{\text{def}}{=} (?A.\mathsf{end}_?) \,\&\, (!(\mathsf{Unit}{+}A).\mathsf{end}_?)$. One client receives an A along r_0, and enqueues it. The other one dequeues an element, and sends it along r_1.

$$
\begin{aligned}
L &\stackrel{\text{def}}{=} \mathsf{empty} \\
M_{enq} &\stackrel{\text{def}}{=} \mathsf{let}\ (v, y'') \;=\; \mathsf{recv}\ y'\ \mathsf{in} \\
&\quad\ \mathsf{let}\ \_ \;=\; \mathsf{terminate}\ y''\ \mathsf{in}\ \mathsf{enq}(z, v) \\
M_{deq} &\stackrel{\text{def}}{=} \mathsf{let}\ (v, z') \;=\; \mathsf{deq}\ z\ \mathsf{in} \\
&\quad\ \mathsf{let}\ \_ \;=\; \mathsf{terminate}\ (\mathsf{send}\ v\ y')\ \mathsf{in}\ z' \\
M &\stackrel{\text{def}}{=} \mathsf{case}\ y\ \mathsf{of}\ y'.\{M_{enq}, M_{deq}\}
\end{aligned}
\qquad
\begin{aligned}
C_0 &\stackrel{\text{def}}{=} \mathsf{let}\ (v, r_0') \;=\; \mathsf{recv}\ r_0\ \mathsf{in} \\
&\quad\ \mathsf{let}\ \_ \;=\; \mathsf{terminate}\ r_0'\ \mathsf{in} \\
&\quad\ \mathsf{let}\ x_0' \;=\; \mathsf{send}\ v\ (\mathsf{select}_L\ x_0)\ \mathsf{in}\ x_0' \\
C_1 &\stackrel{\text{def}}{=} \mathsf{let}\ (v, x_1') \;=\; \mathsf{recv}\ (\mathsf{select}_R\ x_0)\ \mathsf{in} \\
&\quad\ \mathsf{let}\ \_ \;=\; \mathsf{terminate}\ (\mathsf{send}\ v\ r1)\ \mathsf{in}\ x_1' \\
\mathsf{clients} &\stackrel{\text{def}}{=} \mathsf{let}\ x \;=\; \mathsf{fork}_x\ x_0.\, C_0\ \mathsf{in} \\
&\quad\ \mathsf{let}\ x \;=\; \mathsf{fork}_x\ x_1.\, C_1\ \mathsf{in}\ \mathsf{eof}_x
\end{aligned}
$$

We then define $\mathsf{server} \stackrel{\text{def}}{=} \mathsf{serve}\ y\{L, z.\,M, f.\,f\}$, and see that

$$r_0 : {?A}.\mathsf{end}_?, r_1 : {!(\mathsf{Unit} + A)}.\mathsf{end}_? \vdash \mathsf{connect}(x.\,\mathsf{clients}; y.\,\mathsf{server}) : T$$

5.3 Nondeterminism

Unsurprisingly, the races in our system suffice to implement nondeterministic choice. We define $\mathbb{B} \stackrel{\text{def}}{=} \mathsf{Unit} + \mathsf{Unit}$. We implement tt and ff by the obvious injections, and the conditional by

$$\frac{\Gamma \vdash B : \mathbb{B} \qquad \Delta \vdash M : V \qquad \Delta \vdash N : V}{\Gamma, \Delta \vdash \mathsf{if}\ B\ \mathsf{then}\ M\ \mathsf{else}\ N \stackrel{\text{def}}{=} \mathsf{match}\ B\ \mathsf{with}\ x.\{M, N\} : V}\ {+E}$$

(for x fresh). The clients C_0, C_1 respectively send ff and tt over a channel. We also define a server with a pair of Booleans as internal state. The first component records whether the server has ever received a value. When a value is received it is stored in the second component, and any further values received are discarded.

$$
\begin{aligned}
C_0 &\stackrel{\text{def}}{=} \mathsf{send}\ \mathsf{ff}\ x_0 \\
C_1 &\stackrel{\text{def}}{=} \mathsf{send}\ \mathsf{tt}\ x_1 \\
\mathsf{clients} &\stackrel{\text{def}}{=} \mathsf{let}\ x \;=\; \mathsf{fork}_x\ x_0.\, C_0\ \mathsf{in} \\
&\quad\ \mathsf{let}\ x \;=\; \mathsf{fork}_x\ x_1.\, C_1\ \mathsf{in} \\
&\quad\ \mathsf{eof}_x
\end{aligned}
\qquad
\begin{aligned}
M &\stackrel{\text{def}}{=} \mathsf{let}\ (z_0, z_1) \;=\; z\ \mathsf{in} \\
&\quad\ \mathsf{let}\ (v, y') \;=\; \mathsf{recv}\ y\ \mathsf{in} \\
&\quad\ \mathsf{let}\ \_ \;=\; \mathsf{terminate}\ y'\ \mathsf{in} \\
&\quad\ \mathsf{if}\ z_0\ \mathsf{then}\ z\ \mathsf{else}\ (\mathsf{tt}, v) \\
N &\stackrel{\text{def}}{=} \mathsf{let}\ (f_0, f_1) \;=\; f\ \mathsf{in}\ f_1
\end{aligned}
$$

We define a $\mathsf{server} \stackrel{\text{def}}{=} \mathsf{serve}\ y\{(\mathsf{ff}, \mathsf{ff}), z.\,M, f.\,N\}$ beginning from $(\mathsf{ff}, \mathsf{ff})$. We then have that

$$\vdash \mathsf{flip} \stackrel{\text{def}}{=} \mathsf{connect}(x.\,\mathsf{clients}; y.\,\mathsf{server}) : \mathbb{B}$$

This program is translated to $[\![\mathsf{flip}]\!]_y \vdash y : \mathbf{2}$, with reactions $[\![\mathsf{flip}]\!]_y \longrightarrow^* \mathsf{ff}_y$ and $[\![\mathsf{flip}]\!]_y \longrightarrow^* \mathsf{tt}_y$. We can use this to implement a nondeterministic choice operator:

$$\frac{P \vdash \Gamma \qquad Q \vdash \Gamma}{\mathsf{choose}(P, Q) \stackrel{\text{def}}{=} \nu xy.\,([\![\mathsf{flip}]\!]_y \mid \mathsf{if}(x;\ P;\ Q)) \vdash \Gamma}$$

such that $\mathsf{choose}(P, Q) \longrightarrow^* P$ and $\mathsf{choose}(P, Q) \longrightarrow^* Q$.

5.4 Fork–Join Parallelism

Fork-join parallelism [Conway 1963] is a common model of parallelism in which child processes are *forked* to perform computation simultanously. Once they have finished, they are *joined* by the parent process, which collects their work and produces the final result. We assume a 'heavyweight' function $h : A \to B$ that will run on forked processes, and a relatively less expensive function $g : B \to B \to B$ that will combine their answers. We also assume an initial value $g_0 : B$, and a list of 'tasks' $xs : [A]$ to process. Of course, $[A]$ is the type of lists of A, and is supported by the operations:

$$\mathsf{nil} : [A] \qquad \mathsf{cons} : A \to [A] \to [A] \qquad \mathsf{fold}_C : C \to (C \to A \to C) \to [A] \to C$$

Let

$$\mathsf{clients} \stackrel{\text{def}}{=} \mathsf{let}\ y\ =\ \mathsf{fold}_{¿T_S}\ c\ (\lambda x.\, \lambda v.\, \mathsf{fork}_x\ x'.\, (\mathsf{let}\ v'\ =\ h\ v\ \mathsf{in}\ \mathsf{send}\ v'\ x'))\ xs\ \mathsf{in}\ \mathsf{eof}_y$$

$$M \stackrel{\text{def}}{=} \mathsf{let}\ (v, y')\ =\ \mathsf{recv}\ y\ \mathsf{in}\ \mathsf{let}\ \_\ =\ \mathsf{terminate}\ y'\ \mathsf{in}\ g\ z\ v$$

The client protocol is $T_S \stackrel{\text{def}}{=} !B.\mathsf{end}_!$. To form the client pool, we begin with a shared client channel $c : ¿T_S$. We fold over the list $xs : [A]$, adding a forked process for each 'task' $v : A$ to the client pool. Each one of these forked processes will compute $h\ v : B$, and send it over its fresh channel $x' : T_S$. We have $c : ¿T_S \vdash \mathsf{clients} : \mathsf{end}_!$.

We let $\mathsf{server} \stackrel{\text{def}}{=} \mathsf{serve'}\ y\{g_0, z.\, M, f.\, f\}$. The server begins with internal state $g_0 : B$. It nondeterministically receives the result of a computation of h from each client, and 'merges' it into its state using g. In the end, it returns the result. We have $z : B, y : \overline{T_S} \vdash M : B$, and thus $y : ¡\overline{T_S} \vdash \mathsf{server} : B$. We use serve' to pass unlimited parameters to the server internals.

Putting this system together, we get

$$\vdash \mathsf{fork\text{-}join}(h, g_0, g, xs) \stackrel{\text{def}}{=} \mathsf{connect}(x.\, \mathsf{clients}; y.\, \mathsf{server}) : B$$

The fork-join paradigm is often used in industrial parallelization frameworks [Blumofe et al. 1995; Dagum and Menon 1998; Leijen et al. 2009; Reinders 2007]. The background languages and type systems usually do not use any logical devices for concurrency. In particular, concurrent behaviour is not controlled by the type system, as it is here. Note that fork-join requires each spawned process to be independent of each other and only communicate with the parent process, which is precisely caputured by the linearity restriction of our system.

Another parallel computation model is that of *async-finish*. It is more expressive than fork-join, as it allows spawned processes to spawn further processes. The whole tree is then joined at the root process, with no regard to the spawning thread of each child. Our system(s) does not support that: in the ReqA rule, the spawned process M is only given a channel $x' : T_S$, which cannot be used to spawn further processes in the same pool. However, it is well-known is that nested parallelism is still possible, but each child has to spawn its own instance of a fork-join computation, which does not interfere with the root process.

An even more expressive model is that of *futures* [Halstead 1984]. A future is a first-class value that represents a computation running in parallel to the current process. At any point it can be *forced* to obtain its result; if it has not finished an error may be returned, or the process forcing it may block. While fork-join or async-finish spawned processes are independent of each other, futures may be passed around freely (in any reasonably expressive language) and introduce rich interactions. This seems to be in violation of the linearity restriction of our system(s), and thus cannot be expressed. Nevertheless, the Conn rule can be seen as a very restricted form of future, where the spawned process can only communicate with the parent process. More discussions about the difference between these models is given by Acar [2016].

5.5 Keynes' Beauty Contest

Until this point we have seen only relatively simple examples of client-server interaction. In all cases, the 'internal server protocol' we have used has consisted of an unlimited type, the values of which we can replicate or discard. This leads to the false impression that clients access the server one-by-one in a sequential manner, so that clients that connect later are unable to influence the information observed by the earlier ones. In this section we present an example that shows this to be untrue. In particular, if the 'internal server protocol' consists of a session type itself, then we witness bidirectional, interleaving behaviour. This distinguishes our systems from those based on *manifest sharing* [Balzer and Pfenning 2017].

We present a server implementing the umpire in a *Keynesian beauty contest* [Keynes 1936, §12]. Keynes' beauty contest works as follows. A newspaper runs a beauty contest in which readers have to pick the prettiest faces from a set of photographs. The competitors are not those pictured, but the readers themselves: if they pick the faces which are judged to be the prettiest by the majority, they will win a prize. Thus, the readers are incentivized to estimate the aesthetics of the majority.

We will implement a restricted version of this scenario, where a pool of clients votes for a Boolean value. The server then counts the votes, and awards a payoff of 0 or 1 (represented by ff and tt respectively) to each client, indicating whether they voted for the winner. This is obviously impossible if the server handles requests sequentially. In fact, the server will be implemented by spawning a network of interconnected processes, each of which will handle one vote.

We first define the following derived rule. Informally, this rule expresses that a process that uses a channel of type T_S is also exposing a channel of dual type $\overline{T_S}$.

$$\frac{\Gamma, x : T_S \vdash M : \mathsf{end}_! \qquad y : \overline{T_S} \vdash y : \overline{T_S}}{\Gamma \vdash \mathsf{inv}_x(M) \stackrel{\text{def}}{=} \mathsf{connect}(x.\, M; y.\, y) : \overline{T_S}}$$

The client session type is $C_S \stackrel{\text{def}}{=} !\mathbb{B}.?\mathbb{B}.\mathsf{end}_!$, and the internal server protocol is $T_S \stackrel{\text{def}}{=} ?(\mathbb{N} \otimes \mathbb{N}).!\mathbb{B}.\mathsf{end}_?$, where $\mathbb{N}$ is the type of natural numbers. We assume a bunch of standard functions:

$$\mathsf{zero} : \mathbb{N} \qquad \mathsf{succ} : \mathbb{N} \to \mathbb{N} \qquad \leq\, : \mathbb{N} \to \mathbb{N} \to \mathbb{B} \qquad \mathsf{eq} : \mathbb{B} \to \mathbb{B} \to \mathbb{B}$$

where eq checks Boolean values for equality. We let

$$\begin{aligned}
L \stackrel{\text{def}}{=}\ & \mathsf{let}\ w' = \mathsf{send}\ (\mathsf{zero}, \mathsf{zero})\ w\ \mathsf{in} && \text{(send initial state)}\\
& \mathsf{let}\ (_, w'') = \mathsf{recv}\ w'\ \mathsf{in}\ w'' && \text{(receive final value)}\\
N \stackrel{\text{def}}{=}\ & \mathsf{let}\ (s, f') = \mathsf{recv}\ f\ \mathsf{in} && \text{(receive final count)}\\
& \mathsf{let}\ (n_0, n_1) = s\ \mathsf{in} && \text{(unpack state)}\\
& \mathsf{let}\ f'' = \mathsf{send}\ (n_0 \leq n_1)\ f'\ \mathsf{in} && \text{(compute winner and notify the last worker process)}\\
& \mathsf{terminate}\ f'' && \text{(close channel)}\\
M \stackrel{\text{def}}{=}\ & \mathsf{let}\ (s, z') = \mathsf{recv}\ z\ \mathsf{in} && \text{(get state)}\\
& \mathsf{let}\ (n_t, n_f) = s\ \mathsf{in} && \text{(unpack state)}\\
& \mathsf{let}\ (b, y') = \mathsf{recv}\ y\ \mathsf{in} && \text{(receive a vote)}\\
& \mathsf{let}\ s' = \mathsf{if}\ b\ \mathsf{then}\ (\mathsf{succ}\ n_t, n_f)\ \mathsf{else}\ (n_t, \mathsf{succ}\ n_f)\ \mathsf{in} && \text{(increment the right counter)}\\
& \mathsf{let}\ w' = \mathsf{send}\ s'\ w\ \mathsf{in} && \text{(pass new state to next worker process)}\\
& \mathsf{let}\ (b', w'') = \mathsf{recv}\ w'\ \mathsf{in} && \text{(receive winner from next worker process)}\\
& \mathsf{let}\ _ = \mathsf{terminate}\ (\mathsf{send}\ (\mathsf{eq}\ b\ b')\ y')\ \mathsf{in} && \text{(tell competitor if they won, close channel)}\\
& \mathsf{let}\ _ = \mathsf{terminate}\ (\mathsf{send}\ b'\ z')\ \mathsf{in} && \text{(forward winner on, close channel)}\\
& w''
\end{aligned}$$

We define server $\stackrel{\text{def}}{=}$ serve $y\{\mathsf{inv}_w(L), z.\,\mathsf{inv}_w(M), f.\,N\}$. The components are typed as

$$w : \overline{T_S} \vdash L : \mathsf{end}_! \qquad f : T_S \vdash N : \mathsf{unit}$$
$$w : \overline{T_S}, z : T_S, y : \overline{C_S} \vdash M : \mathsf{end}_! \qquad y : ¡T_S \vdash \mathsf{server} : \mathsf{unit}$$

The details of this protocol are subtle. The construct $\mathsf{inv}_x(-)$ allows us to use programs which only have side-effects as internal server state, by inverting the polarity of one of the channels. The server is initialized by L, which sets the state to be $(0, 0)$. It then listens on the same channel to receive the winner, which it promptly discards. The server finalization N receives the final tally of the votes, computes the winner, sends back the result, and closes the channel.

The component M is used to communicate with each competitor. It receives the state of the server, the competitor's vote, and increments the appropriate tally. It then passes on this new state to the next worker process M, which will communicate with the next competitor. This sets up an entire network of worker processes M, one to serve each competitor. When the competitors have all cast their votes, N computes the winner, and sends it back to the last worker process. This process then tells the competitor whether they won, closes the channel to the competitor, and passes on the result to the worker process serving the previous competitor, and so on. At the very end, the winner is passed to the initialization process L.

We can then define a number of competitors $x_i : !\mathbb{B}.?\mathbb{B}.\mathsf{end}_! \vdash C_i : \mathsf{end}_!$ who will cast their votes by sending a Boolean value and receive a payoff along x_i. These can be combined into a client pool, much in the same way as in previous examples.

If we have two such competitors C_0 and C_1 merged in a pool, and we connect them to server, we will obtain a process topology of the form illustrated in the schematic diagram of Fig. 7. Compared with Fig. 4 this diagram is intuitive but loose on accuracy. Details such as $\mathsf{end}_?$ and $\mathsf{end}_!$ are left out. We have also spelled out the protocols internals. For example, the server internal protocol T_S is indicated by a forward arrow $\mathbb{N} \otimes \mathbb{N}$ and a backward arrow $\mathbb{B}$.

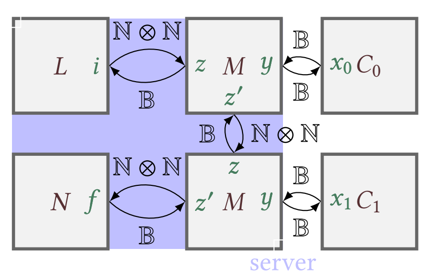

Fig. 7. Layout of the Keynesian beauty contest, after co-exponential reactions but before other reactions. Boxes represents processes whose names are at the center of the boxes. Arrows represents directed mesages between processes with types of the data annotated. Labels on edges of boxes are the names of the channels to the processes.

6 RELATED WORK

Hypersequents and Session Types. Hypersequents were introduced to process calculi and Classical Linear Logic by Montesi and Peressotti [2018]. Another version of that system was studied in detail by Kokke et al. [2019a]. A reaction semantics similar to the one used here was given in [Kokke et al. 2018].

Differential Linear Logic. The rules for ¿ given in §2.2 are almost the same as the *coweakening*, *codereliction* and *cocontraction* rules for ! in Differential Linear Logic (DiLL) [Ehrhard 2018]. DiLL is equipped with nondeterministic reduction and formal sums, and is believed to have something to do with concurrency. Ehrhard and Laurent [2010] have produced an embedding of the finitary π-calculus into DiLL, though that encoding has been criticized [Mazza 2018]. A type of client-server interactions—namely the encoding of ML-style reference cells into session types—has been encoded by Castellan et al. [2020] in a system based on the rules of DiLL. This work relies on both the costructural rules and Mix, so it is not clear which device primarily augments expressive power.

Our work shows that something akin to the costructural rules of DiLL arises from the wish to form client pools. The exact relationship between coexponentials and DiLL remains to be determined.

Clients, Servers, and Races in Linear Logic. Typing client-server interaction has been a thorn in the side of session types and Linear Logic. All previous attempts rely on some version of the Mix rule. Both Wadler [2014, §3.4] and Caires and Pérez [2017, Ex. 2.4] use Mix to combine clients into client pools. Kokke et al. implicitly use Mix to type an otherwise untypable client pool in HCP [Kokke et al. 2019a, Ex. 3.7].[7] Remarkably, none of these calculi demonstrate stateful server behaviour, as we predicted using a semantic argument in §1.1.

Atkey et al. [2016] explore the additional power bestowed upon CP by *conflating* dual connectives. The conflation of ? and ! leads to the notion of *access point*, a dynamic match-making communication service on a single end point. In fact, the rules look eerily close to the list-like formulation of our servers and generators. Access points prove too powerful: they introduce stateful nondeterminism, racy communication, and general recursion. This impairs the safety of CP by introducing deadlock and livelock. Our work shows that we can still safely obtain the former two features without introducing the third.

Adding nondeterminism to CLL in a controlled fashion is complex. Atkey et al. [2016] express a form of *nondeterministic local choice* in CP by conflating $\&$ and $\oplus$. The resultant form of nondeterministic choice cannot induce the racy behaviour normally exhibited in the π-calculus [Kokke et al. 2019b, §2]. Caires and Pérez [2017] present a dual-context system based on CLL+Mix in which the same kind of nondeterministic local choice is expressed through a new set of modalities, $\oplus$ and $\&$.[8] These bear a similarity to the coexponential modalities presented here, but they are used for nondeterminism instead. Their $\&$ modality has a monadic flavour, and hence can be used to encapsulate nondeterminism 'in the monad' in the usual manner in which we isolate effects.

Kokke et al. [2019b] drew inspiration from Bounded Linear Logic [Girard et al. 1992] to formulate a system for nondeterministic client-server interaction. They use types of the form $?_n A$ (standing for n copies of A delimited by $\parr$) and $!_n A$ (standing for n copies of A delimited by $\otimes$). $!_n A$ represents a pool of n disjoint clients with protocol A, and $?_n A$ a server that can serve exactly n clients with protocol A. While this is consistent with disjoint-vs.-connected concurrency, their system is limited to serving a specific number of clients in each session. Thus, it fails to satisfy criterion (i) in §1.1, and does not form a satisfactory model.

Carbone et al. [2017] approach multiparty session types through *coherence proofs*. In *op. cit.* the authors develop *Multiparty Classical Processes*, a version of CP with role annotations and the *MCut* rule. The latter is a version of the MultiCut rule annotated with a *coherence* judgment derived from Honda et al. [2016], which generalises duality and ensures that roles match appropriately. MCP does not allow dynamic sessions with arbitrary numbers of participants, and hence cannot model client-server interactions. MCP was later refined into the system of Globally-governed Classical Processes (GCP) by Carbone et al. [2016]. Unlike these calculi, our work does not require any consideration of coherence or local vs. global types.

The work of Rocha and Caires [2021], which also appears at ICFP 2021, introduces shared state to CLL. This is accomplished by a rule akin to the co-contraction of DiLL [Ehrhard 2018], along with rules that manipulate shared memory cells (allocation, deallocation, read, write). Races are resolved on a case-by-case basis, using locks to protect critical sections, and collecting all the possible outcomes into a formal sum, again in the style of DiLL. This system also includes the Mix rule, but it is not clear if that is an essential feature of the approach. Moreover, the system enjoys subject reduction, progress, weak normalization, and—as nondeterminism is captured by formal sums—it

[7] This has been confirmed to us by the authors.
[8] This is an intentional clash with external and internal choice in Linear Logic.

is also confluent. Compared to that work the approach through the coexponential modalities is more parsimonious and follows Linear Logic more closely. The difference in expressivity is unclear.

Fixed Points in Linear Logic. Inductive and coinductive types—presented proof-theoretically as least and greatest fixed points—were introduced in the context of higher-order Classical Linear Logic by Baelde [2012]. Baelde formulates a weakly normalization cut elimination procedure, which albeit does not satisfy the subformula property. Ehrhard and Jafarrahmani [2021] study categorical models for a slight extension of the propositional fragment of Baelde's system, which allow them to infer certain facts about the behaviour of (co)inductive linear data types.

The structure of Baelde's system has been used to extend CP with inductive and coinductive types by Lindley and Morris [2016]. This is our starting point in §2.2, but with significant alterations along the way. First, our system is based on HCP. Second, the server rule has been reformulated to support hyperenvironments, and server finalization (§3.3). Third, our client pools allow permutation in order to enable nondeterminism (§3.4). Finally, our reaction semantics are tailored to the specific setting, and are consequently simpler. In a separate strand of work, Toninho et al. [2014] introduce coinduction in a system of session types based on Intuitionistic Linear Logic (ILL); see Lindley and Morris [2016, §§1, 7] for a comparison. Derakhshan and Pfenning [2020] give a linear metalogic with least and greatest fixed point and prove strong progress for binary session-typed processes in the metalogic.

Manifest sharing. Closely related to our work is the notion of *manifest sharing* [Balzer and Pfenning 2017]. This work starts from a very different premise: a channel is either *linear* (as the usual channels in session types), or *shared* between processes. This leads to an ILL-based system, SILL_S, with two *modes* and two modalities shifting between them [Reed 2009]. The switch to a shared channel is punctuated by the modalities. Thus, sharing *manifests* in the types. In some ways, SILL_S is a much stronger system, as it features *equi-synchronizing* recursive session types. The price to pay is the introduction of deadlock. Balzer et al. [2019] develop an additional layer of the type system that protects from it.

Our work attempts to solve the expressivity problem of LL-based session types beginning from Curry-Howard: we seek the minimal extension to Linear Logic that will enable us to write server and client processes. Unlike manifest sharing, we remain committed to CLL and its duality. The result is that our system has simpler rules, avoiding the notions of linear and shared channels (as (co)exponentials internalise them), as well as the lock-like primitives used to introduce modalities by Balzer and Pfenning [2017]. Moreover, we have remained committed to the goal of retaining the good properties ensured by cut elimination in CLL (e.g. deadlock freedom). A drawback of this approach is that our system inherits the linearity constraint from linear logic, and is thus unable to express circular structures (such as Dijkstra's dining philosophers).

Both systems provide atomicity, but in radically different ways. In SILL_S users access the service in a mutually exclusive manner. This is not compatible with the usual view of typical client-server interaction, where mutliple clients need to access the server simultanously in order to exchange information. A common workaround is to decompose an interleaved session into a 'stateless' protocol consisting of several mini-sessions (such as the voting examples in Das et al. [2021] and Sano et al. [2021]). Every client is then required to send a 'cookie' to identify themselves across mini-sessions. In our system accesses to the shared service are concurrent, but causally atomic (§4). As a result, interleaved sessions can be expressed natively (§5.5).

Type systems for the π-calculus. There are many ways to equip the π-calculus with a type system. A large class of such systems is based on Kobayashi's notion of *channel usage* [Kobayashi 2002, 2003, 2006]. That work proceeds in the opposite direction: it begins with the π-calculus, and tries to tame its expressive power through types that control the use of channels, thereby guaranteeing deadlock-freedom, lock-freedom, and so on. These systems are able to express some of the expected properties of client-server interaction, see e.g. Kobayashi [2003, Example 8]. Comparing these families of type systems for concurrent behaviour is a difficult task which has been undertaken by Dardha and Pérez [2015]. The main difference seems to be that our work tries to stick as closely as possible to the foundations of session types in linear logic. In addition, the usage-based type systems take a 'channel-first' approach, where all channels may be shared between processes; this is in sharp contrast to session types [Kobayashi 2003, §10]. Dardha and Gay [2018] have attempted to merge these two approaches through the formulation of Priority-based CP, a new calculus based on CLL which allows a controlled form of cyclic dependencies.

Session Types. There is a nontrivial connection between our work and *Multiparty Session Types* [Coppo et al. 2016; Honda et al. 2008, 2016], which comprise a π-calculus and a behavioural type system specifying interaction between multiple agents. The kinds of protocols expressed by multiparty session types are 'fully' choreographed, and involve a *fixed* number of participants. As such, they cannot model interactions with an arbitrary number of clients; nor can they introduce a controlled amount of nondeterminism. Some of these expressive limitations have been remedied in systems of *Dynamic Multirole Session Types* [Deniélou and Yoshida 2011], which come at the price of introducing *roles* that parties can dynamically join or leave, and a notion of quantification over participants with a role. Our system captures certain use-cases of roles using only tools from linear logic, with little additional complexity.

7 CONCLUSIONS AND FURTHER WORK

We presented the system of Client-Server Linear Logic, which features a novel form of modality, the coexponentials. We then showed how CSLL can be used to model client-server interactions without falling down the slippery slope of introducing Mix. We comment on some directions for future work.

Termination. It would be interesting to establish a *termination* result for CSLL. This would prove that the resulting calculus does not generate *livelock*. We expect this proof to be somewhat involved, which is why most work on Linear Logic and session types either fails to produce a proof, or defers to Girard's proof for CLL [Aschieri and Genco 2019; Wadler 2014].

Syntax. The weak ¡ rule listed in §2.2 is expressed by folding $\otimes$ over the set of formulas. This obstructs a particular commuting conversion in cut elimination. Similarly, presentation of the *strong* exponential and its computational interpretation is omitted due to its unsatisfactory rules. We believe these issues are due to the limitation of sequent calculus, and alternative techniques are necessary to solve them.

ACKNOWLEDGEMENTS

Alex Kavvos was supported in part by a research grant (no. 12386, Guarded Homotopy Type Theory) from the VILLUM Foundation. This work was also supported in part by a Villum Investigator grant (no. 25804, Center for Basic Research in Program Verification (CPV)). We gratefully acknowledge discussions with Martin Berger, Amar Hadzihasanovic, Vladimir Zamdzhiev, Wen Kokke, Fabrizio Montesi and Marco Peressotti.

REFERENCES

Samson Abramsky. 1993a. Computational interpretations of linear logic. *Theoretical Computer Science* 111, 1-2 (1993), 3–57. https://doi.org/10.1016/0304-3975(93)90181-R ISBN: 0304-3975.

Samson Abramsky. 1993b. Interaction categories. In *Theory and Formal Methods 1993*. Springer, 57–69. https://doi.org/10.1007/978-1-4471-3503-6_5

Samson Abramsky. 1994. Proofs as processes. *Theoretical Computer Science* 135, 1 (1994), 5–9. https://doi.org/10.1016/0304-3975(94)00103-0

Samson Abramsky, Simon J Gay, and Rajagopal Nagarajan. 1996. Interaction categories and the foundations of typed concurrent programming. In *Deductive Program Design (Nato ASI Subseries F)*, Manfred Broy (Ed.). Springer-Verlag Berlin Heidelberg, 35–113. http://www.springer.com/us/book/9783540609476

Samson Abramsky and Radha Jagadeesan. 1994. Games and Full Completeness for Multiplicative Linear Logic. *The Journal of Symbolic Logic* 59, 2 (1994), 543. https://doi.org/10.2307/2275407 arXiv:1311.6057

Umut A Acar. 2016. *Parallel Computing: Theory and Practice*. http://www.cs.cmu.edu/afs/cs/academic/class/15210-f15/www/tapp.html

Federico Aschieri and Francesco A. Genco. 2019. Par Means Parallel: Multiplicative Linear Logic Proofs as Concurrent Functional Programs. *Proc. ACM Program. Lang.* 4, POPL, Article 18 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371086

Robert Atkey. 2017. Observed communication semantics for classical processes. In *European Symposium on Programming*. Springer, 56–82. https://doi.org/10.1007/978-3-662-54434-1_3

Robert Atkey, Sam Lindley, and J. Garrett Morris. 2016. Conflation Confers Concurrency. In *A List of Successes That Can Change the World*, Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella (Eds.). Lecture Notes in Computer Science, Vol. 9600. Springer International Publishing, 32–55. https://doi.org/10.1007/978-3-319-30936-1_2

Arnon Avron. 1991. Hypersequents, logical consequence and intermediate logics for concurrency. *Annals of Mathematics and Artificial Intelligence* 4, 3-4 (1991), 225–248. https://doi.org/10.1007/BF01531058

David Baelde. 2012. Least and greatest fixed points in linear logic. *ACM Transactions on Computational Logic* 13, 1 (2012), 1–44. https://doi.org/10.1145/2071368.2071370

Stephanie Balzer and Frank Pfenning. 2017. Manifest sharing with session types. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–29. https://doi.org/10.1145/3110281

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *Programming Languages and Systems*, Luís Caires (Ed.), Vol. 11423. Springer International Publishing, Cham, 611–639. https://doi.org/10.1007/978-3-030-17184-1_22

Michael Barr. 1991. $*$-Autonomous categories and linear logic. *Mathematical Structures in Computer Science* 1, 2 (1991), 159–178. https://doi.org/10.1017/S0960129500001274

Gianluigi Bellin. 1997. Subnets of proof-nets in multiplicative linear logic with MIX. *Mathematical Structures in Computer Science* 7, 6 (1997), 663–669. https://doi.org/10.1017/S0960129597002326

G. Bellin and P. J. Scott. 1994. On the π-calculus and linear logic. *Theoretical Computer Science* 135, 1 (1994), 11–65. https://doi.org/10.1016/0304-3975(94)00104-9

N Benton and P Wadler. 1996. Linear logic, monads and the lambda calculus. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE. https://doi.org/10.1109/LICS.1996.561458

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Santa Barbara, California, USA) *(PPOPP '95)*. Association for Computing Machinery, New York, NY, USA, 207–216. https://doi.org/10.1145/209936.209958

Luís Caires and Jorge A. Pérez. 2017. Linearity, Control Effects, and Behavioral Types. In *Programming Languages and Systems. ESOP 2017*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 229–259. https://doi.org/10.1007/978-3-662-54434-1_9

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *Proceedings of the 21st International Conference on Concurrency Theory* (Paris, France) *(CONCUR'10)*. Springer-Verlag, Berlin, Heidelberg, 222–236. https://doi.org/10.5555/1887654.1887670

Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear logic propositions as session types. *Mathematical Structures in Computer Science* 26, 3 (2016), 367–423. https://doi.org/10.1017/S0960129514000218

Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. 2016. Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types. In *27th International Conference on Concurrency Theory (CONCUR 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 59)*, Josée Desharnais and Radha Jagadeesan (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 33:1–33:15. https://doi.org/10.4230/LIPIcs.CONCUR.2016.33

Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. 2017. Multiparty session types as coherence proofs. *Acta Informatica* 54 (2017), 243–269. https://doi.org/10.1007/s00236-016-0285-y

Simon Castellan, Léo Stefanesco, and Nobuko Yoshida. 2020. Game Semantics: Easy as Pi. *CoRR* abs/2011.05248 (2020). arXiv:2011.05248

Melvin E. Conway. 1963. A Multiprocessor System Design *(AFIPS '63 (Fall))*. Association for Computing Machinery, New York, NY, USA, 139–146. https://doi.org/10.1145/1463822.1463838

Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science* 26, 2 (2016), 238–302. https://doi.org/10.1017/S0960129514000188

L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55. https://doi.org/10.1109/99.660313

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 (Lecture Notes in Computer Science, Vol. 10803)*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, 91–109. https://doi.org/10.1007/978-3-319-89366-2{_}5

Ornela Dardha and Jorge A. Pérez. 2015. Comparing Deadlock-Free Session Typed Processes. *Electronic Proceedings in Theoretical Computer Science* 190 (2015). https://doi.org/10.4204/EPTCS.190.1

A. Das, S. Balzer, J. Hoffmann, F. Pfenning, and I. Santurkar. 2021. Resource-Aware Session Types for Digital Contracts. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, Los Alamitos, CA, USA, 111–126. https://doi.org/10.1109/CSF51468.2021.00004

Pierre-Malo Deniélou and Nobuko Yoshida. 2011. Dynamic Multirole Session Types. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '11*. ACM Press, 435. https://doi.org/10.1145/1926385.1926435

Farzaneh Derakhshan and Frank Pfenning. 2020. Circular Proofs in First-Order Linear Logic with Least and Greatest Fixed Points. (2020). arXiv:2001.05132

Thomas Ehrhard. 2018. An introduction to differential linear logic: proof-nets, models and antiderivatives. *Mathematical Structures in Computer Science* 28, 7 (2018), 995–1060. https://doi.org/10.1017/S0960129516000372

Thomas Ehrhard and Farzad Jafarrahmani. 2021. Categorical models of Linear Logic with fixed points of formulas. arXiv:2011.10209 To appear in the proceedings of LICS 2021.

Thomas Ehrhard and Olivier Laurent. 2010. Interpreting a finitary pi-calculus in differential interaction nets. *Information and Computation* 208, 6 (2010), 606–633. https://doi.org/10.1016/j.ic.2009.06.005

Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional asynchronous session types: session types without tiers. *Proceedings of the ACM on Programming Languages* 3, POPL (2019). https://doi.org/10.1145/3290341

Simon J Gay and Vasco T Vasconcelos. 2010. Linear type theory for asynchronous session types. *Journal of Functional Programming* 20, 1 (2010), 19. https://doi.org/10.1017/S0956796809990268

Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. https://doi.org/10.1016/0304-3975(87)90045-4

J. Y. Girard and Y. Lafont. 1987. Linear logic and lazy computation. In *TAPSOFT '87 (Lecture Notes in Computer Science, Vol. 250)*, Hartmut Ehrig, Robert Kowalski, Giorgio Levi, and Ugo Montanari (Eds.). Springer-Verlag, Berlin/Heidelberg, 52–66. https://doi.org/10.1007/BFb0014972

Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. 1992. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science* 97, 1 (1992), 1–66. https://doi.org/10.1016/0304-3975(92)90386-T

Robert H. Halstead. 1984. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, USA) *(LFP '84)*. Association for Computing Machinery, New York, NY, USA, 9–17. https://doi.org/10.1145/800055.802017

Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming* (revised first ed.). Morgan Kaufmann. https://doi.org/10.5555/2385452

Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems: Proceedings of the 7th European Symposium on Programming (ESOP'98) (Lecture Notes in Computer Science, Vol. 1381)*. Springer, Berlin, Heidelberg, 122–138. https://doi.org/10.1007/BFb0053567

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press. https://doi.org/10.1145/1328438.1328472

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 1–67. https://doi.org/10.1145/2827695

John Maynard Keynes. 1936. *The General Theory of Employment, Interest and Money*. Macmillan & Co. Ltd., London.

Naoki Kobayashi. 2002. A Type System for Lock-Free Processes. *Information and Computation* 177, 2 (2002), 122–159. https://doi.org/10.1006/inco.2002.3171

Naoki Kobayashi. 2003. Type Systems for Concurrent Programs. In *Formal Methods at the Crossroads. From Panacea to Foundational Support (Lecture Notes in Computer Science, Vol. 2757)*, Bernhard K. Aichernig and Tom Maibaum (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 439–453. https://doi.org/10.1007/978-3-540-40007-3_26 Extended version.

Naoki Kobayashi. 2006. A new type system for deadlock-free processes. In *International Conference on Concurrency Theory*. Springer, 233–247. https://doi.org/10.1007/11817949_16

Wen Kokke, Fabrizio Montesi, and Marco Peressotti. 2018. Taking Linear Logic Apart. In *Proceedings Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity-TLLA@FLoC 2018, Oxford, UK, 7-8 July 2018 (EPTCS, Vol. 292)*, Thomas Ehrhard, Maribel Fernández, Valeria de Paiva, and Lorenzo Tortora de Falco (Eds.). 90–103. https://doi.org/10.4204/EPTCS.292.5

Wen Kokke, Fabrizio Montesi, and Marco Peressotti. 2019a. Better late than never: a fully-abstract semantics for classical processes. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29. https://doi.org/10.1145/3290337

Wen Kokke, J Garrett Morris, and Philip Wadler. 2019b. Towards races in linear logic. In *International Conference on Coordination Languages and Models*. Springer, 37–53. https://doi.org/10.1007/978-3-030-22397-7_3

Yves Lafont and Thomas Streicher. 1991. Games semantics for linear logic. In *Proceedings 1991 Sixth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 43–44. https://doi.org/10.1109/LICS.1991.151629

Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. 2009. The design of a task parallel library. *Acm Sigplan Notices* 44, 10 (2009), 227–242. https://doi.org/10.1145/1639949.1640106

Sam Lindley and J Garrett Morris. 2015. A semantics for propositions as sessions. In *European Symposium on Programming Languages and Systems*. Springer, 560–584. https://doi.org/10.1007/978-3-662-46669-8_23

Sam Lindley and J. Garrett Morris. 2016. Talking bananas: structural recursion for session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming - ICFP 2016*. ACM Press, Nara, Japan, 434–447. https://doi.org/10.1145/2951913.2951921

Sam Lindley and J. Garrett Morris. 2017. Lightweight Functional Session Types. In *Behavioural Types: from Theory to Tools*, Simon Gay and Antonio Ravara (Eds.). River Publishers. https://doi.org/10.13052/rp-9788793519817

J. Maraist, M. Odersky, D.N. Turner, and P. Wadler. 1999. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Theoretical Computer Science* 228, 1-2 (1999), 175–210. https://doi.org/10.1016/S0304-3975(98)00358-2

John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. 1995. Call-by-name, Call-by-value, Call-by-need, and the Linear Lambda Calculus. *Electronic Notes in Theoretical Computer Science* 1 (1995), 370–392. https://doi.org/10.1016/S1571-0661(04)00022-2

Damiano Mazza. 2018. The true concurrency of differential interaction nets. *Mathematical Structures in Computer Science* 28, 7 (2018), 1097–1125. https://doi.org/10.1017/S0960129516000402

Paul-André Melliès. 2009. Categorical Semantics of Linear Logic. In *Panoramas et synthèses 27: Interactive models of computation and program behaviour*, Pierre-Louis Curien, Hugo Herbelin, Jean-Louis Krivine, and Paul-André Melliès (Eds.). Société Mathématique de France. http://www.pps.univ-paris-diderot.fr/~mellies/papers/panorama.pdf

Paul-André Melliès, Nicolas Tabareau, and Christine Tasson. 2018. An explicit formula for the free exponential modality of linear logic. *Mathematical Structures in Computer Science* 28, 7 (2018). https://doi.org/10.1017/S0960129516000426

Massimo Merro and Davide Sangiorgi. 2004. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science* 14, 5 (2004), 715–767. https://doi.org/10.1017/S0960129504004323

Robin Milner. 1992. Functions as processes. *Mathematical Structures in Computer Science* 2, 2 (1992), 119–141. https://doi.org/10.1017/S0960129500001407

Robin Milner. 1999. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, New York, NY, USA. https://doi.org/10.5555/329902

Robin Milner, Joachim Parrow, and David Walker. 1992. A calculus of mobile processes, i. *Information and computation* 100, 1 (1992), 1–40. https://doi.org/10.1016/0890-5401(92)90008-4

Fabrizio Montesi and Marco Peressotti. 2018. Classical Transitions. (2018). arXiv:1803.01049

Jason Reed. 2009. A Judgmental Deconstruction of Modal Logic. (2009). http://www.cs.cmu.edu/~jcreed/papers/jdml.pdf

James Reinders. 2007. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.".

Pedro Rocha and Luís Caires. 2021. Propositions-as-Types and Shared State. *Proceedings of the ACM on Programming Languages* ICFP (2021). https://doi.org/10.1145/3473584

Chuta Sano, Stephanie Balzer, and Frank Pfenning. 2021. Manifestly Phased Communication via Shared Session Types. *CoRR* abs/2101.06249 (2021). arXiv:2101.06249 https://arxiv.org/abs/2101.06249

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. https://doi.org/10.1007/978-3-662-45917-1_11

Maarten van Steen and Andrew S. Tanenbaum. 2017. *Distributed Systems* (3 ed.). distributed-systems.net. https://www.distributed-systems.net/

Vasco T. Vasconcelos. 2012. Fundamentals of session types. *Information and Computation* 217 (2012), 52–70. https://doi.org/10.1016/j.ic.2012.05.002

Philip Wadler. 2014. Propositions as sessions. *Journal of Functional Programming* 24, 2-3 (2014), 384–418. https://doi.org/10.1017/S095679681400001X