Concurrency And Races In Classical Linear Logic

Zesen Qian

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Concurrency And Races In Classical Linear Logic

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Zesen Qian
August 30, 2022

Abstract

Recent works have extended the successful Proposition-As-Types correspondence to concurrent computing in the form of a tight correspondence between linear logic and process calculus. In particular, linear logic propositions are equated with session types. Process calculi based on the tight correspondence derive from linear logic good properties such as deadlock-freedom and session fidelity. On the other hand, however, those calculi are usually poor in expressivity. They are usually deterministic and lack common concurrency features such as threads and shared states, and can thus barely model real-world concurrency. Following works proposed systems with stronger expressivity but usually at the cost of deviating from the correspondence and introducing bad behaviors. The present thesis, on the other hand, seeks non-determinism within the framework of linear logic, which can be introduced to the system without compromising the logical structure and thus the good properties. We found two such examples of non-determinism of interest. One is between clients and a server, where clients race to be accepted by the server. The other one is between effectful computations and an effect handler, where multiple computations race to emit effects to be dealt with by the handler. We formulate two extensions based on the two examples. Our extensions are further strengthened by least and greatest fixed points so that variable quantities can be supported, including variable numbers of clients, variable numbers of effects in a computation, and variable numbers of racing computations. Good properties such as deadlock-freedom are proved, and several examples are given to demonstrate the expressivity of our extensions.

Resumé

I denne afhandling foreslår og studerer vi en ny korrespondence mellem lineær logik og nye type systemer for process kalkuler, der modellerer programmeringssprog med parallelle beregninger. De nye type systemer indfanger flere vigtige features i process kalkuler, herunder ikke-determinisme og effekter, og den tætte korrespondence med liner logik betyder, at type systemerne giver nogle gode garantier for hvorledes veltypede process udtryk opfører sig, når de bliver evalueret. For eksempel garanteres det at evaluering ikke vil lede til deadlock.

Acknowledgments

First and foremost, I want to thank my supervisor Lars Birkedal for his guidance on not only research but also soft skills throughout my PhD studies, skills that will be of great value whether in academia or industry. I also want to extend a big thanks to Alex Kavvos for a long and fruitful collaboration. Although Alex did not officially supervise me, he provided numerous advice and guidance, and deeply influenced my PhD studies. I also want to thank Fabrizio Montesi and Marco Peressotti for hosting me at SDU and the successful collaboration.

I also want to extend my gratitude to the people that helped and encouraged me during my early studies. In particular, I want to thank Yubin Xia for introducing me into computer systems, and Steve Awodey and Anders Mörtberg for introducing me into the mathematical side of type theory.

Finally, a big thanks goes to my friends and family who have supported me during my studies. I thank you all for the casual chats we had over lunch, coffee and zoom — they were vital distractions to ride out difficult times.

Zesen Qian,
Aarhus, August 30, 2022.

Contents

Chapter 1

Overview

1.1 A Brief History of Proofs as Processes

The Early Days Ever since its inception, linear logic [Girard, 1987a] was believed to have deep connections with concurrency. This was motivated by the comparison with intuitionistic logic. Recall that in the latter, judgements have the form

$$\Gamma \vdash A$$

where Γ is the set of assumptions and A is the conclusion; they are not symmetric and are obviously directional: the information flows from the assumptions to the conclusion. As a result, in usual computational interpretations, a proof of $\Gamma \vdash A$ is interpreted as a program where Γ is inputs and A is output. In addition, the cut rule

$$\frac{\Gamma \vdash A \qquad \Delta, A \vdash B}{\Gamma, \Delta \vdash B}$$

is interpreted as using the output of the first program as the input of the second program.

In contrast, judgements in linear logic has the form

$$\vdash A_0, \cdots, A_n$$

which is single sided, and all formulas A_i are symmetric, in the sense that there is no distinguished member. The corresponding cut rule is

$$\frac{\vdash \Gamma, A \qquad \vdash \Delta, A^{\perp}}{\vdash \Gamma, \Delta}$$

Note that in linear logic we have $A^{\perp\perp} = A$, and thus A and $A^{\perp}$ are symmetric; therefore, the two premises are symmetric. One can still rewrite the two premises to

$$\vdash \Gamma^{\perp} \multimap A \qquad\qquad \vdash A \multimap \Delta$$

and interpret the first as a program converting $\Gamma^{\perp}$ to A, and the second as a program converting A to Δ. However, this is certainly less than ideal as it ruins the nice symmetry of linear logic. A natural interpretation that reflects the symmetry is to be found.

Girard [1987b, 1989] suggested a connection between linear logic and concurrency in the same style of the hugely successful connection between intuitionistic logic and functional programming (i.e. Proposition-As-Types). The idea was made concrete by Abramsky [1994] and Bellin and Scott [1994], who established the first interpretation of proofs as processes in the π-calculus [Milner et al., 1992]. The details are as follows. First of all, formulas in sequents are given names:

$$\vdash x_0 : A_0, \cdots, x_n : A_n$$

which induces named versions of proof rules. For example the named cut rule would be

$$\frac{\vdash \Gamma, x : A \qquad \vdash \Delta, y : A^{\perp}}{\vdash \Gamma, \Delta}$$

The names are useful for the next step, which is to *inductively* intrepret each linear logic (named) proof as a π-calculus term, where the free names of the proof matches the free names of the term. Therefore, propositions map to sessions, and derivations of proofs map to construction of processes. In particular, the above named cut rule is interpreted as putting P and Q in parallel and connecting x and y, where P and Q are the processes of the two premises. Note that the free names of a π-process term are symmetric: there is no 'distinguished' port for 'output'; also note that when interpreting the cut rule P and Q are treated equally. The new interpretation seems to reflect the most distinctive feature of linear logic in comparison to intuitionistic logic.

The new interpretation, however, only uses a significantly small subset of π-calculus terms. In particular, $A \otimes B$ maps to sending a pair of types A and B, and $A \parr B$ maps to receiving. This is a severe restriction on canonical π-calculus, where processes can send and receive any terms. Moreover, this restriction can hardly be motivated from a logical perspective. In summary, the correspondence is not as tight as desired.

The Twist An attempt at the previous problem came sixteen years later when Caires and Pfenning [2010] discovered that intuitionistic linear propositions can be interpreted as session types [Honda, 1993, Honda et al., 1998]. $A \otimes B$ is interpreted as sending A and continuing as B, while $A \parr B$ as receiving A and continuing as B. The system (called πDILL) is based on *internal* π-calculus [Sangiorgi, 1996] which is though just as expressive [Boreale, 1998] as canonical π-calculus. In internal π-calculus, processes cannot send any names freely, but only freshly generated ones. In some sense, the sending operation is the binder of the fresh name whose scope is the continuation. To understand, consider the reduction rules for sending and receiving in canonical π-calculus and in internal π-calculus respectively:

$$\nu z\,(z(x).\,P \mid z[y].\,Q) \to \nu z\,(P[y/x] \mid Q) \qquad \text{(canonical)}$$
$$\nu z\,(z(x).\,P \mid z[y].\,Q) \to \nu z\,\nu y\,(P[y/x] \mid Q) \qquad \text{(internal)}$$

In canonical π-calculus, $z[y]$ sends exising y over z, which is then received by $z(x)$. In particular, y is free on both sides. In internal π-calculus, $z[y]$ binds the name y and sends it over x which is received by $z(x)$. In particular, y is bound on both sides: on the left by $z[y]$ and on the right by νy. The restriction of sending and receiving pairs are apparently removed at the cost of deviation from canonical π-calculus.

The work has still some shortcomings. First of all, while connections to session types is made, the system is based on intuitionistic linear logic which is a constrained version of classical linear logic and lacks duality; it is similar to the relation between intuitionistic logic an classical logic. As a result, the duality between session types has no correspondence in their system. Duality is a central element of session types, and the lack of it severely undermines the attempts to relate session types.

Wadler [2014] solves the issue by adapting the correspondence to classical linear logic. With the duality recovered, linear logic types now tightly corresponds to session types, and also the system is more concice and symmetric. The system is called Classical Process (CP) and will be the base for most following works. The system (and its derivatives) is so minimalistic, however, that the user is left to code the basic building blocks, which becomes cumbersome quickly. To remedy, the author introduced a higher-level language inspired by Gay and Vasconcelos [2010]. Its semantics is given as translations to CP, from which nice properties would be easily inherited. In addition to common building blocks such as functional programming, it also contains explicit session programming backed by the interpretation of linear logic types as session types. For example, one can send and receive over sessions:

$$\vdash \mathsf{send} : A \multimap (!A.B) \multimap B \qquad\qquad \vdash \mathsf{recv} : ?A.B \multimap A \otimes B$$

send takes two arguments, the datum to send typed A, and the session over which the datum will be send. The seesion is typed $!A.B$ the session can be used to send A and will continue as B; indeed, B is the return type of *send*. *recv* takes a session which is typed $?A.B$ meaning the session can be used to receive a datum typed A and will continue as B, which is confirmed by the return type of *recv* which is a pair of the newly received datum and the rest of the session.

CP is also more tightly connected to linear logic compared to previous works. Both Bellin and Scott [1994] and Caires and Pfenning [2010] translate linear logic to π-calculus, and correspondingly cut-elimination to reduction. However, as Wadler [2014] observed this is achieved at costs. The first issue is about the axiom rule, which is intuitively a bidirectional forwarder between two sessions of dual types. Bellin and Scott [1994] restricted the type to be atoms, because forwarding composite sessions would require an ad-hoc complex processes. Indeed, this was implemented in Caires and Pfenning [2010, prop 4.2]. Wadler [2014] solved this issue by giving the axiom rule a special and uniform semantics: if a session x of process P communicates with the session y of the forwarder $y \leftrightarrow z$, then all occurances of x in P are replaced by z.

Another issue is that of commuting conversions. In canonical π-calculus, a process is understood as a sequence of actions that need to fire in order. However, cut elimination of linear logic when translated to π-calculus involves sometimes earlier actions in a process firing later. Caires and Pfenning [2010] forbids those commuting conversions to stay close to π-calculus while deviates from linear logic; Wadler [2014] on the other hand 'let linear logic guide the design of process calculus', and adopted those commuting conversions. For example consider the following rule:

$$\nu z\,(x[y].\,(P \mid Q) \mid R) \to x[y].\,(\nu z\,(P \mid R) \mid Q)$$

where P and R communicates via z. In canonical π-calculus, the left hand side would just block, because there does not exist another process that receives on x. This is however not acceptable in CP, because the term corresponds to a proof with top-level cut, and proofs should satisfy cut-elimination. Therefore, we must make the left hand side reduce, and in the way that corresponds to how cuts are eliminated. To that end, the above rule is introduced to CP corresponding to a rule in linear logic.

The reason why commuting conversion is needed in linear logic, is that all reduction rules have a form where the reacting actions are immediately under the restriction. In the above rule we see on the left hand side P clearly blocked by $x[y]$, and thus cannot use the reduction rules. The rule solves the issue by moving the cut lower so that it restricts both and only P and R. Note that the first actions of P and R might still not be on z, in which case more commuting conversions will be performed. This happens repeatedly until they meet the requirement of a reduction rule, upon which two dual actions on z will react. Note that the action $x[y]$ stays put and not fired, giving the impression that this action, while appears earlier in the process, fires later. This might seem strange for π-calculus people; however, Wadler [2014] pointed out that the close connections to linear logic immediately gives progress and termination of CP, just like how that of simply typed lambda calculus follows immediately from intuitionistic logic.

A New Era In addition to actions firing out-of-order, there were a few other important mismatches between processes and linear logic remaining in the syntax and semantics. Most notably, parallel composition had no direct correspondence to a rule in the sequent calculus of linear logic. Carbone et al. [2018] proposed that parallel composition corresponds to composing hypersequents [Avron, 1991] which are collections of sequents. In contrast to previous works, in this setting each process might have multiple disjoint components, corresponding to the multiple sequents in its hypersequents. As a result, composing processes corresponds to composing hypersequents:

$$\frac{P \vdash \mathcal{G} \qquad Q \vdash \mathcal{H}}{P \mid Q \vdash \mathcal{G} \mid \mathcal{H}} \qquad\qquad \frac{}{\mathbf{0} \vdash \emptyset}$$

The rule on the right is the nullary version giving the empty process typed by empty hypersequents just for completeness. Some rules such as cut can be thought as having parallel composition baked-in, and can now be decoupled using hypersequents:

$$\frac{P \vdash \Gamma, x : A \mid \Delta, y : A^{\perp}}{\nu xy\, P \vdash \Gamma, \Delta}$$

Hypersequents further enabled the reconstruction of the expected labelled transition system (LTS) semantics [Montesi and Peressotti, 2018], an explanation of the hypersequent-based approach in linear logic [Kokke et al., 2019a, 2018], and the expected metatheoretical results of session types (session fidelity) and bisimilarity [Montesi and Peressotti, 2021].

LTS can be thought as a decomposition of reduction semantics. One observes that reduction always happens between two dual actions. For example we have the following reduction rule in CP:

$$\nu xy\, (x(x').\, P \mid y[y'].\, Q) \to \nu xy\, \nu x'y'\, (P \mid Q)$$

which specifies the reaction between sending ($y[y']$) and receiving ($x(x')$). LTS decomposes it into three rules:

$$x(x').\, P \xrightarrow{x(x')} P \qquad\qquad y[y'].\, Q \xrightarrow{y[y']} Q \qquad\qquad \frac{R \xrightarrow{x(x')\mid y[y']} R'}{\nu xy\, R \xrightarrow{\tau} \nu xy\, \nu x'y'\, R'}$$

where the first rule says that a process starting with the action $x(x')$ would just signal the label $x(x')$ and transition into the continuation; similar for $y[y']$. The first two rules concerns only actions and are thus called *action rules*, while the third rule bridge them together and are called *communication rule*, which we now explain. In the premise, we require a process R to be able to simultanously signal both sending and receiving labels, which means it must contain at least two processes ready to send and receive. One such R would be $x(x').\, P \mid y[y'].\, Q$. The conclusion says that if we connect x

and y of R, the two processes would be able to communicate; moreover, the empty label τ indicates that the communication is not observable externally, as the sending and receiving signal 'offset' each other. One can derive the reduction semantics from the LTS semantics, if we think of $\xrightarrow{\tau}$ as reduction.

LTS has several benefits over reduction. First of all, of course, it makes available the existing toolbox with regards to behaviours such as bisimilarity. Secondly, as a decomposition of reduction it is finer-grained and better-behaved in many scenarios; for example, commuting conversion is no longer needed. Recall the above process $\nu z\,(x[y].\,(P \mid Q) \mid R)$ which requires commuting conversion; in LTS it can simply transition as

$$\nu z\,(x[y].\,(P \mid Q) \mid R) \xrightarrow{x[y]} \nu z\,(P \mid Q \mid R)$$

Note that the $x[y]$ action simply fires and signals a label, and does not block the reaction between P and R. Similarly, LTS is capable of specifying the semantics of primitve effects whereas reduction cannot (chapter 3).

Another benefit of LTS is that it gives a natural notion of well-behaved programs. Recall that a program that reduces infinitely is considered bad-behaved; however, there are also programs such as servers that reduces infinitely but produces information for each step, which is considered well-behaved. The distinction is usually hard to formulate, but in LTS well-behaved processes can simply be formulated as those that keep signaling labels until it becomes the empty process $\mathbf{0}$, with one exception: it should not keep signaling τ-labels infinitely as that produces no information.

The problem The deep connections with linear logic ensure that the resulting concurrent systems enjoy good properties such as deadlock-freedom, session-fidelity and livelock-freedom. However, all the systems mentioned are 'too well-behaved' to model the chaotic nature of real world concurrent systems. To be exact, they are deterministic and in particular missing common concurrency features such as thread, shared states and locks.

1.2 Proposals

The present thesis attempts to increase the expressivity of Classical Process systems without compromising their good behavioral properties derived from the close connections to linear logic. To that end, we formulate concurrency primitives in the framework of linear logic, so that concurrency-related transitions correspond to logical implications. To better understand the point, we need to establish some intuition about linear logic connectives. Following are (slightly modified) rules for $\otimes$ and $\parr$:

$$\otimes\ \frac{\vdash \Gamma, A_0 \mid \Delta, A_1}{\vdash \Gamma, \Delta, A_0 \otimes A_1} \qquad\qquad \parr\ \frac{\vdash \Sigma, A_0^\perp, A_1^\perp}{\vdash \Sigma, A_0^\perp \parr A_1^\perp}$$

For $\otimes$, in the premise A_0 and A_1 come from two sequents; recall that sequents in a hypersequent are disjoint from each other. For the dual rule, $A_0^\perp$ and $A_1^\perp$ come from the same sequents and are thus connected.

Server/Client One could cut $A_0 \otimes A_1$ and $A_0^\perp \parr A_1^\perp$. While A_0 and A_1 are not connected directly, they are now respectively connected to $A_0^\perp$ and $A_1^\perp$ which are connected and are therefore connected indirectly now. We find this phenomenon nicely models the interaction between a client pool and a server. One can think A_0 and A_1 to be the interfaces of two clients respectively, and $A_0^\perp \parr A_1^\perp$ to be the interfaces of the corresponding server that serves the two clients. Moreover, the analysis beforehand means that the clients are connected not directly, but indirectly via the server; this nicely fits our intuition about server/client interactions. Note that we should think A_0 and A_1 to be the same type with the subscription only for identification, which reflects that in real life server often provides the same service to several clients. We now apply the server/client interpretation to the logical implication:

$$A_0 \otimes A_1 \to A_1 \otimes A_0$$

and find that it swaps the ordering of two clients. Moreover, the implication is bidirectional, meaning the swapping is reversible. Note either ordering will cut with $A_0^\perp \parr A_1^\perp$ as A_0 and A_1 are the same type. This nicely models one aspect of non-determinism of client/server interaction; namely, clients race to get accepted by the server. The idea is developed in detail in chapter 2.

Effects Consider the two dual propositions (right associative):

$$A \otimes B^\perp \parr C \qquad\qquad A^\perp \parr B \otimes C^\perp$$

In the first proposition, A is disjoint from both $B^\perp$ and C, the two of which are connected. Vice versa for the second proposition. We find this nicely models the interaction between an effectful computation and a handler, where A is a request, B is a response, and C is the return value of the effectful computation. Most notably, the request A is disjoint from the continuation $B \parr C$, as there is no causation between them; however, once we cut the two together, the request A will be connected to the continuation indirectly via the handler. That means there is causation between request and continuation only via the handler. Apply the interpretation to the following logical implication:

$$(A_0 \otimes {B^\perp}_0 \parr C_0) \otimes (A_1 \otimes {B^\perp}_1 \parr C_1) \to A_0 \otimes {B^\perp}_0 \parr (C_0 \otimes (A_1 \otimes {B^\perp}_1 \parr C_1))$$

and we find that it propagates the effect of the first computation out of two parallel computations. There is a symmetric implication that propagates the effects of the second computation. Together the two rules allow racing for effects among two parallel computations. Moreover, both implications are unidirectional, meaning that the outcome of the race cannot be reversed. This idea is developed in detail in chapter 3.

Fixed Points In both server/client and effects, there are several desired generalizations in regards to quantity. In the former, one hopes that a client pool can contain any number of clients. In the latter, one hopes that an effectful computation that can emit effects any number of times; moreover, any number of effectful computations should be able to race in parallel. The apparent infinite nature of the generalized settings leads us to the concept of least and greatest fixed points, which is ubiquitous in functional programming languages and corresponds to inductive and coinductive *data*. They are introduced to Classical Process by Lindley and Morris [2016] and correspond to inductive and coinductive *sessions*. Inductive sessions are finite, while coinductive sessions are potentially infinite. The two are specified to be dual and will communicate; moreover, the communication terminates because the inductive side is finite and terminates, forcing the coinductive side to terminate as well. Our works are based on theirs, but with significant alterations along the way. First, our extensions are based on hypersequents. Second, our rules and their semantics are specialized to the particular functors, and therefore simpler. Thirdly, our semantics allow non-determinism to better model concurrency, while being logically equivalent to the original fixed points and preserving nice properties of the base system.

1.3 Related Works

In this section we discuss related works in general; those more particular to our work are discussed in the corresponding chapters.

Expressivity in Classical Processes Based on CP, GV Wadler [2014] as well as later systems [Fowler et al., 2019, Lindley and Morris, 2015] allows forking of processes. However, the child process runs in isolation with a single channel connected to the parent. This is essentially a Cut and does not increase expressivity.

Lindley and Morris [2016] introduces inductive and coinductive sessions to CP. It is adapted from Baelde [2012], which introduces least and greatest fixed points to classical linear logic. A categorical semantics is given by Ehrhard and Jafarrahmani [2021]. All mentioned works including ours on least and greatest fixed point break the subformula property, as the greatest fixed point requires an internal state which could be arbitrary type and generally not a subformula of the greatest fixed point type. In another strand of work, Toninho et al. [2014] introduces 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] gives a linear metalogic with least and greatest fixed point and proves strong progress for binary session-typed processes in the metalogic.

In addition to least and greatest fixed points, there are other ways to allow variable quantity when seeking extra expressivity. Most related works to be discussed below introduced new types with their own rules for specific use cases, and do not strictly correspond to any least and greatest fixed points. Some of them are not well-behaved, but it is unclear to what degree quantity variability contributes to that, as the systems are often coupled with non-determinism which brings chaos. In any case, in our opinion, quantity variability is not the central feature of concurrency, and we should focus on other aspects in the following discussion of related works.

Atkey et al. [2016] explores obtaining more power in CP by *conflating* dual connectives. Conflating $\&$ and $\oplus$ gives local non-deterministic choices which however cannot induce the racy behavior normally exhibited in the π-calculus [Kokke et al., 2019b, §2]. Conflating ? and ! gives to the notion of *access point*, a dynamic match-making communication service on a single endpoint. The rules look eerily close to the list-like formulation of our servers. 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 works show that we can still safely obtain the former two features without introducing the third.

Carbone et al. [2017] introduces the standard local non-determinism into linear logic. Caires and Pérez [2017] presents 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 $\&$.[1] These bear a similarity to the coexponential modalities presented in chapter 2, but they are used for nondeterminism instead. Their $\&$ modality has a monadic flavor, and hence can be used to encapsulate nondeterminism 'in the monad' in the usual manner in which we isolate effects.

Carbone et al. [2017] approaches multiparty session types through *coherence proofs*. 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 generalizes duality and ensures that roles match appro-

[1]This is an intentional clash with external and internal choice in Linear Logic.

priately. 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 works do not require any consideration of coherence or local vs. global types.

Manifest sharing Closely related to our work is the notion of *manifest sharing* [Balzer and Pfenning, 2017]. Their system is stratified into two layers, *linear* and *shared*, where the former behaves as ILL and the latter as IL. The sharing manifests in the types, in that one can tell the layer from the look of each type. Two modalities shift between the two layers [Reed, 2009] and are computationally interpreted as *acquire* and *release*. Similar to the common notion of locking, the two operations are blocking and might cause deadlocks, in particular when there are circular dependencies. Balzer et al. [2019] adds priorities to types to prevent circularity and thus recover deadlock-freedom.

Among the two extensions of ours, manifest sharing is probably closer to effects, as both allow multiple sequential accesses to the shared state. Our works attempt to solve the expressivity problem of LL-based session types beginning from Curry-Howard: we seek the minimal extension to linear logic that models client/server and effects. Unlike manifest sharing, we remain committed to CLL and its duality. As a result, our systems have simpler rules, avoid the notions of linear and shared channels, and avoid 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) without unsafe extension.

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 [2003, 2002, 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 can 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 behavior 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* [Honda et al., 2008, 2016, Coppo et al., 2016], which comprise a π-calculus and a behavioral 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 non-determinism. 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 systems capture certain use-cases of roles using only tools from linear logic, with little additional complexity.

1.4 Future works

Termination and Readiness It would be interesting to establish a *termination* result for CSLL (chapter 2). This would prove that the resulting calculi do 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 proof or defers to Girard's proof for CLL Wadler [2014], Aschieri and Genco [2019]. Similarly, we want to establish *readiness* for CELL (chapter 3) without retry (section 3.5) by extending the proof in Montesi and Peressotti [2021]. Readiness can be understood as productivity and is more general than termination. This should not be hard, considering that the difficult part which is the quantity variability already exists in their system (namely, the exponentials).

Syntax The weak $¡$ rule listed in section 2.2.2 is expressed by folding $\otimes$ over the set of formulas. It is less 'native' than the weak exponential rule where folding $\parr$ can be simply represented as commas. And indeed, this big $\otimes$ obstructs a particular commuting conversion in cut elimination. Similarly, the presentation of the *strong* exponential and its computational interpretation is omitted due to its unsatisfactory rules. There are several other occurrences where sequent calculus syntax obstructs.

We believe these issues are due to the limitation of sequent calculus; in particular, a sequent is understood as a collection of formulas $\parr$-together, and a hypersequent is understood as a collection of sequent $\otimes$-together. $\otimes$

and $\parr$ are not treated equally, although they should be dual. New syntax frameworks such as deep inference [Tubella and Straßburger, 2019] promised a better formulation, but our preliminary trials demand a deeper investigation.

Additive Units All languages in the CP family omit the additive units ($\mathbf{0}$ and $\top$); we believe they can model crashes (unrecoverable errors). Consider the rule for $\top$:

$$\frac{}{\top, \Gamma}\ \textsc{Top}$$

Or more intuitively: $\mathbf{0} \multimap \Gamma$. One can view the above process as superficially exposing $\Gamma, \top$ while being void internally. What happens when one tries to communicate to the Γ? This is given by the commutative case of cut elimination:

$$\frac{\dfrac{}{\top, A, \Gamma} \qquad A^{\perp}, \Delta}{\Gamma, \Delta, \top} \quad \to \quad \frac{}{\Gamma, \Delta, \top}$$

We see that the crash propagates: $\Delta, A^{\perp}$ which used to have real content, is not voided as well. We have an empty process still but with bigger types.

The fault $\top$ once introduced cannot be hidden away: there is no rule for $\mathbf{0}$ to cut with $\top$; the only way to acquire $\mathbf{0}$ is to use Ax rule, in which case we will just introduce another $\top$. In terms of programming, that means one can 'handle faults', but the handler cannot suppress the faults: it has to be propagated. For example, we can have a type of term $(\top \oplus 2) \multimap (\top \oplus 2)$ which takes a possibly faulty boolean, and negate it; the result is again a possibly faulty boolean. However, we can not have a term of type $(\top \oplus 2) \multimap 2$ which suppresses the fault.

Guarded Recursion Greatest fixed points are potentially infinite but do not compromise readiness because they are always ready to produce a labeled transition. (Co)induction is however a rather regulated form of recursion, and one sometimes needs a more liberated one similar to general recursion (self-reference) in functional programming. Recursive types, as well as recursive processes, are formulated in Montesi and Peressotti [2021], while unsurprisingly breaking readiness. A possible future exploration is to borrow ideas from guarded recursion in functional programming [Birkedal et al., 2017] to linear logic, so one can define a wider range of programs with recursion while preserving readiness.

Chapter 2

Client-Server Sessions in Linear Logic

This chapter is based on Qian et al. [2021], but I removed part of section 2.1 that is general to classical processes, which has been incorporated into section 1.1. I also removed part of section 2.6 that is general to classical processes, which has been incorporated into section 1.3. I also reformatted the text to fit the new paper size.

2.1 Introduction

2.1.1 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 [1987a]. 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:

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

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 $w[\text{USE}].\, xw[\text{USE}].\, yP \vdash w : ?A$. 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:

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

Mix was carefully considered for inclusion in Linear Logic, but was rejected [Girard, 1987a, §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:

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

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

$$\textsc{Mix0}\quad \dfrac{}{\mathbf{0} \vdash \cdot}$$

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:

$$\dfrac{\dfrac{}{\mathbf{0} \vdash \cdot}\,\textsc{Mix0}}{\mathbf{0} \vdash x : ?A}\,?w$$

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 Bellin [1997], Girard [1987a], Abramsky et al. [1996], Wadler [2014], Atkey et al. [2016] that Mix is equivalent to

$$A \otimes B \multimap A \parr B \qquad (*)$$

where $C \multimap D := C^\perp \parr D$.

Admitting this implication is unwise. At first glance, (2.1) 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, 1987a, §V.4]) and $A \otimes B$ as *disjoint concurrency* (no information flow between A and B whatsoever). The implication (2.1) 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, (2.1) 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 $\parr$, 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 $\parr$. As we cannot cut a $\parr$ (in the server) with another $\parr$ (in the client pool), we are compelled to also accept the converse implication $A \parr B \multimap A \otimes B$ in order to convert one of the two $\parr$'s to $\otimes$. This forces $\otimes = \parr$, which inescapably leads to deadlock [Atkey et al., 2016, §4.2].

Requiring $\otimes = \parr$, a.k.a. *compact closure* Barr [1991], Abramsky et al. [1996], is often deemed necessary for concurrency. In fact, Atkey et al. [2016] argue that this *conflation of dual connectives* ($\mathbf{1} = \perp$, $\otimes = \parr$, 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]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.

2.1.2 Roadmap

First, in section 2.2 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 section 2.3 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 section 2.4 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 section 2.5 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 section 2.6.

2.2 Exponentials, Fixed Points, and Coexponentials

2.2.1 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

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

one defines

$$!A := \nu F_A \qquad\qquad ?A := \mu G_A$$

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 $(A \to B)^o := !A^o \multimap B^o$ 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

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

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

where $A^{\otimes n} := A \otimes \cdots \otimes A$, 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.

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.

2.2.2 Deriving Coexponentials

Both exponentials (qua fixed points) are given by a tree where each fork is marked with a connective ($\otimes$ for !, $⅋$ 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 $⅋$). Let

$$H_A(\mathcal{X}) := \bot \,\&\, A \,\&\, (\mathcal{X} ⅋ \mathcal{X}) \qquad K_A(\mathcal{X}) := \mathbf{1} \oplus A \oplus (\mathcal{X} \otimes \mathcal{X})$$

The strong coexponentials are then defined by

$$¡A := \nu H_A \qquad ¿A := \mu K_A$$

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

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

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

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.

2.2.3 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 section 2.1.1) secretly introduces the coexponential modalities proposed here.

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

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

BiCut cuts two formulas at once, and MultiCut an arbitrary number. These rules were first proposed in the context of Linear Logic by Abramsky [1993a] 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 $C \multimap D := C^\perp \parr D$. Some form of the following lemma may be found across the relevant literature Girard [1987a], Abramsky et al. [1996], Bellin [1997], Wadler [2014], Atkey et al. [2016].

Lemma 1. *The following rules are logically interderivable.*

(i) The axiom $\mathbf{1} \multimap \perp$ *and the* Mix0 *rule.*

(ii) The axiom $\perp \multimap \mathbf{1}$ *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* $\vdash A^\perp, A$ *and* BiCut*.*

Armed with this, we can prove that:

Theorem 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 [Wadler, 2014, Kokke et al., 2019a]. 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.

2.3 Processes

In the rest of the paper we will argue that the logical observations we made in section 2.2 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 (section 2.3.1). Following that, we briefly discuss three technical design decisions that pertain to the coexponentials used in our system (section 2.3.2,section 2.3.3,section 2.3.4). Finally, we introduce the system in section 2.3.5, and its metatheory in section 2.3.6.

2.3.1 ¿ Means Client, ¡ Means Server

Recall the three rules for ¿, namely

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

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.

2.3.2 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 section 2.2.1: 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.

2.3.3 Design Decision #2: Replacing Trees with Lists

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

$$H_A(\mathcal{X}) := \perp \mathbin{\&} A \mathbin{\&} (\mathcal{X} \parr \mathcal{X})$$

As discussed in section 2.2.1 and section 2.2.2, this rule represents a *tree-like structure*. Nothing stops us from replacing it with a *list-like* structure.[4] We use the functors

$$H'_A(\mathcal{X}) := \perp \mathbin{\&} (A \parr \mathcal{X}) \qquad\qquad K'_A(\mathcal{X}) := \mathbf{1} \oplus (A \otimes \mathcal{X})$$

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.

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

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

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

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

$$\frac{\dfrac{\vdash \Gamma, B \qquad \vdash B^\perp, \Delta}{\vdash \Gamma, \Delta, B \otimes B^\perp} \qquad \dfrac{\dfrac{}{\vdash B^\perp, B}}{\vdash B^\perp ⅋ B, \bot} \qquad \dfrac{\dfrac{\vdash B^\perp, A, B \qquad \dfrac{}{\vdash B^\perp, B}}{\vdash B^\perp, B, B \otimes B^\perp, A}}{\vdash B^\perp ⅋ B, A ⅋ (B \otimes B^\perp)}}{\vdash \Gamma, \Delta, ¡A}$$

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 $\bot$ rule is invertible, so instantiating $\Delta := \bot$ in Server gives back the preceding rule. Hence, these two rules are logically equivalent.

2.3.4 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

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

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

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

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.

2.3.5 Introducing CS::

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 [1987a]—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. 2.1. 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

$$
\begin{array}{lcll}
A, B, \ldots & ::= & \mathbf{1} \mid \bot \mid A \parr B \mid A \otimes B \mid A \oplus B \mid A \,\&\, B \mid ¿A \mid ¡A \mid ?A \mid !A & \\
\Gamma, \Delta, \ldots & ::= & \cdot \mid \Gamma, x : A & \text{(environments)} \\
\mathcal{G}, \mathcal{H}, \ldots & ::= & \emptyset \mid \mathcal{G} \mid \Gamma & \text{(hyperenvironments)} \\
P, Q, \ldots & ::= & \mathbf{0} & \text{(terminated process)} \\
& & \mid x \leftrightarrow y & \text{(link between } x \text{ and } y) \\
& & \mid \nu xy\, P & \text{(connect } x \text{ and } y) \\
& & \mid P \mid Q & \text{(parallel composition)} \\
& & \mid y.\mathsf{case}\{\textsc{l}{:}P, \textsc{r}{:}Q\} & \text{(receive choice over } y) \\
& & \mid y[\textsc{l}].\, P \mid y[\textsc{r}].\, P & \text{(send choice over } y) \\
& & \mid y(x).\, P \mid y[x].\, P & \text{(receive/send } x \text{ over } y) \\
& & \mid y().\, P \mid y[].\, P & \text{(receive/send end-of-session at } y) \\
& & \mid ¿x[].\, P & \text{(create new client interface } x) \\
& & \mid ¿x[y].\, P & \text{(send client interface } y \text{ over } x) \\
& & \mid ¡y\{z', w', y'.\, Q\}(z, w).\, P & \text{(serve over } y) \\
& & \mid x[\textsc{disp}].\, P \mid \;\; x[\textsc{use}].\, yP \mid \;\; x[\textsc{dup}](y_0).\, y_1 P & \\
& & & \text{(weakening, dereliction and contraction)} \\
& & \mid !x\{\vec{y}.\, P\} & \text{(promotion)}
\end{array}
$$

$$
\textsc{HMix0}\ \dfrac{}{\mathbf{0} \vdash \emptyset}
\qquad
\textsc{HMix2}\ \dfrac{P \vdash \mathcal{G} \qquad Q \vdash \mathcal{H}}{P \mid Q \vdash \mathcal{G} \mid \mathcal{H}}
\qquad
\textsc{Cut}\ \dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : A^\bot}{\nu xy\, P \vdash \mathcal{G} \mid \Gamma, \Delta}
$$

$$
\textsc{Ax}\ \dfrac{}{x \leftrightarrow y \vdash x : A^\bot, y : A}
\qquad
\textsc{Par}\ \dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A, y : B}{y(x).\, P \vdash \mathcal{G} \mid \Gamma, y : A \parr B}
$$

$$
\textsc{Tensor}\ \dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : B}{y[x].\, P \vdash \mathcal{G} \mid \Gamma, \Delta, y : A \otimes B}
\qquad
\textsc{PlusL}\ \dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A}{x[\textsc{l}].\, P \vdash \mathcal{G} \mid \Gamma, x : A \oplus B}
$$

$$
\textsc{PlusR}\ \dfrac{Q \vdash \mathcal{G} \mid \Gamma, y : B}{y[\textsc{r}].\, Q \vdash \mathcal{G} \mid \Gamma, y : A \oplus B}
\qquad
\textsc{With}\ \dfrac{P \vdash \Gamma, x : A \qquad Q \vdash \Gamma, x : B}{x.\mathsf{case}\{\textsc{l}{:}P, \textsc{r}{:}Q\} \vdash \Gamma, x : A \,\&\, B}
$$

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

$$
\textsc{WhyNotD}\ \dfrac{P \vdash \mathcal{G} \mid \Gamma, y : A}{x[\textsc{use}].\, yP \vdash \mathcal{G} \mid \Gamma, x : ?A}
\qquad
\textsc{WhyNotC}\ \dfrac{P \vdash \mathcal{G} \mid \Gamma, y_0 : ?A, y_1 : ?A}{x[\textsc{dup}](y_0).\, y_1 P \vdash \mathcal{G} \mid \Gamma, x : ?A}
$$

$$
\textsc{OfCourse}\ \dfrac{P \vdash \vec{y} : ?\vec{B}, x : A}{!x\{\vec{y}.\, P\} \vdash \vec{y} : ?\vec{B}, x : !A}
\qquad
\textsc{QueW}\ \dfrac{P \vdash \mathcal{G}}{¿x[].\, P \vdash \mathcal{G} \mid x : ¿A}
$$

$$
\textsc{QueA}\ \dfrac{P \vdash \mathcal{G} \mid \Gamma, x : ¿A \mid \Delta, x' : A}{¿x[x'].\, P \vdash \mathcal{G} \mid \Gamma, \Delta, x : ¿A}
$$

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 $|$, 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:

$$\dfrac{\dfrac{P \vdash \Gamma, y : A \qquad Q \vdash \Delta, x : B}{P \mid Q \vdash \Gamma, y : A \mid \Delta, x : B}}{x[y].\, P \mid Q \vdash \Gamma, \Delta, x : A \otimes B} \qquad\qquad \dfrac{\dfrac{}{\mathbf{0} \vdash \emptyset}}{x[].\, \mathbf{0} \vdash x : \mathbf{1}}$$

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 section 2.3.1, section 2.3.2, section 2.3.3,section 2.3.4. 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 $|$ 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 section 2.3.3, but in the interest of brevity it 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

[5] Mix would join them with a comma, which would stand for a $⅋$.

[6] Mix0 would stand for the unit of $⅋$, namely $\perp$.

interfaces are separated by a $\mid$ 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 $x[\textsc{dup}](x_0).\,x_1 P$.
- x is bound in P within $x'[\textsc{use}].\,xP$.

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 $\mathcal{G} := x : A, y : B \mid z : C, w : D$ determines the set of sets $\lfloor \mathcal{G} \rfloor = x, y \mid z, w := \{\{x, y\}, \{z, w\}\}$. Kokke et al. [2019a] call this the *name partition* corresponding to a hyperenvironment. Thus, if $P \vdash \mathcal{G}$ we define $\textsc{Fn}(P) := \lfloor \mathcal{G} \rfloor$. 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.

2.3.6 Operational Semantics and Metatheory

Definition 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.
- $\mathbf{0}$ and $x \leftrightarrow y$ are canonical.

$$P \mid \mathbf{0} \equiv P \qquad (\textsc{Par-Unit})$$

$$P \mid Q \equiv Q \mid P \qquad (\textsc{Par-Comm})$$

$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad (\textsc{Par-Assoc})$$

$$x \leftrightarrow y \equiv y \leftrightarrow x \qquad (\textsc{Link-Comm})$$

$$\nu xy\,(P \mid Q) \equiv P \mid \nu xy\, Q \;\; (x, y \notin \textsc{Fn}(P)) \qquad (\textsc{Res-Par})$$

$$\nu xy\, \nu zw\, P \equiv \nu zw\, \nu xy\, P \qquad (\textsc{Res-Res})$$

$$\pi_x.\,(P \mid Q) \equiv P \mid \pi_x.\,Q \;\; (\textsc{Bn}(\pi_x) \cap \textsc{Fn}(P) = \emptyset) \qquad (\textsc{Pre-Par})$$

$$\nu xy\, \pi_z.\,P \equiv \pi_z.\,\nu xy\, P (z \neq x, y \text{ and } \pi_z \text{ and } \nu xy \text{ not cross } \textsc{Fn}(P)) \qquad (\textsc{Res-Pre})$$

$$\pi_x.\,\pi_y.\,P \equiv \pi_y.\,\pi_x.\,P$$
$$(x \neq y, y \notin \textsc{Bn}(\pi_x), x \notin \textsc{Bn}(\pi_y), \pi_x \text{ and } \pi_y \text{ not cross } \textsc{Fn}(P)) \qquad (\textsc{Pre-Pre})$$

extended with

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

Figure 2.2: The structural equivalence of CSLL processes.

- $y.\mathsf{case}\{\textsc{l}{:}P, \textsc{r}{:}Q\}$ 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 $y.\mathsf{case}\{\textsc{l}{:}P, \textsc{r}{:}Q\}$ 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. 2.2. Furthermore, we define the *reaction relation* $P \longrightarrow Q$ between processes to be the least relation induced by the clauses in fig. 2.3.

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. 2.3.6 and 2.3.6 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' the name partition in order to preserve typing. As a counterexample, if $P \vdash x : A, y : B \mid z : C, w : D$, then $x(w).\,y[z].\,P \vdash x : A \parr D, y : B \otimes C$ while $y[z].\,x(w).\,P$ is ill-typed. To avoid this, we say that $x(w)$ and $y[z]$

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

$$\textsc{Eq}\ \frac{P \equiv P' \qquad P \longrightarrow Q \qquad Q \equiv Q'}{P' \longrightarrow Q'}$$

$$\nu xy\,(z.\mathsf{case}\{\textsc{l}{:}P_0, \textsc{r}{:}P_1\} \mid Q) \longrightarrow z.\mathsf{case}\{\textsc{l}{:}\nu xy\,(P_0 \mid Q), \textsc{r}{:}\nu xy\,(P_1 \mid Q)\} \qquad (\textsc{With-Comm})$$

$$\nu xy\,(!z\{x\vec{w}.\, P\} \mid !y\{\vec{v}.\, Q\}) \longrightarrow !z\{\vec{v}\vec{w}.\, \nu xy\,(P \mid !y\{\vec{v}.\, Q\})\} \qquad (\textsc{OfCourse-Comm})$$

$$\nu xy\,(z \leftrightarrow x \mid Q) \longrightarrow Q[z/y] \qquad (\textsc{Link})$$

$$\nu xy\,(x[].\, P \mid y().\, Q) \longrightarrow P \mid Q \qquad (\textsc{One-Bot})$$

$$\nu xy\,(x[z].\, P \mid y(w).\, Q) \longrightarrow \nu xy\, \nu zw\,(P \mid Q) \qquad (\textsc{Tensor-Par})$$

$$\nu xy\,(x[\textsc{l}].\, P \mid y.\mathsf{case}\{\textsc{l}{:}Q_0, \textsc{r}{:}Q_1\}) \longrightarrow \nu xy\,(P \mid Q_0) \qquad (\textsc{PlusL-With})$$

$$\nu xy\,(x[\textsc{r}].\, P \mid y.\mathsf{case}\{\textsc{l}{:}Q_0, \textsc{r}{:}Q_1\}) \longrightarrow \nu xy\,(P \mid Q_1) \qquad (\textsc{PlusR-With})$$

$$\nu xy\,(¿x[].\, C \mid ¡y\{z, z', y'.\, Q\}(i, f).\, P) \longrightarrow C \mid \nu if\, P \qquad (\textsc{Claro-QueW})$$

$$\nu xy\,(¿x[x'].\, C \mid ¡y\{z, z', y'.\, Q\}(i, f).\, P) \longrightarrow \nu xy\, \nu x'y'\,(C \mid R)$$
$$\text{where} R := ¡y\{z, z', y'.\, Q\}(z', f).\,(\nu iz\,(P \mid Q)) \qquad (\textsc{Claro-QueA})$$

$$\nu xy\,(x[\textsc{disp}].\, P \mid !y\{\vec{z}.\, Q\}) \longrightarrow \vec{z}[\textsc{disp}].\, P \qquad (\textsc{ExpW})$$

$$\nu xy\,(x[\textsc{use}].\, x'P \mid !y\{\vec{z}.\, Q\}) \longrightarrow \nu x'y\,(P \mid Q) \qquad (\textsc{ExpD})$$

$$\nu xy\,(x[\textsc{dup}](x_0).\, x_1 P \mid !y\{\vec{z}.\, Q\}) \longrightarrow \vec{z}[\textsc{dup}](\vec{z_0}).\, \vec{z_1} \nu x_0 y_0\, \nu x_1 y_1\,(P \mid R)$$
$$\text{where } R := !y_1\{\vec{z_1}.\, Q[\vec{z_1} y_1/\vec{z}y]\} \mid !y_0\{\vec{z_0}.\, Q[\vec{z_0} y_0/\vec{z}y]\} \qquad (\textsc{ExpC})$$

Figure 2.3: The operational semantics of CSLL processes.

cross the name partition $\textsc{Fn}(P) = x, y \mid z, w$, and hence that this commutation is forbidden. Formally, 'crossing' is defined as follows.

Definition 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.

$$\begin{aligned} X_{x(x')} &:= \{x, x'\} & X_{x[\textsc{dup}](x_0).\,x_1} &:= \{x_0, x_1\} \\ Y_{y[y']} &:= \{y, y'\} & Y_{¿y[y'].} &:= \{y, y'\} \\ Y_{¡y\{z',w',y'.\,Q\}(z,w).} &:= \{z, w\} \end{aligned}$$

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 $x[\textsc{disp}]$
  - 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 2.3.6 allows us to commute the position of two clients in the pool, thereby imitating racing—as discussed in section 2.3.4. In order fully exploit the nondeterminism induced by 2.3.6 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 2.3.6 first. Indeed, this is the major motivation for 2.3.6, as the latter is not needed for our metatheoretic results. Note that 2.3.6 is the one and only source of nondeterminism in the system.

Some commuting conversions appear as structural equivalences, and some as reaction rules. 2.3.6 and 2.3.6 are commuting conversions for OfCourse and With respectively. 2.3.6 with 2.3.6 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 2.3.6 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 2.3.6, which allows us to swap any two non-interfering 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 section 2.4. 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 2.3.6 corresponds to serving an empty client pool. In this case we simply connect the initialization and finalization channels of P. Likewise, the rule 2.3.6 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 section 2.5.5.

We have the following metatheoretic results.

Lemma 3. *If $P \equiv Q$, then $P \vdash \mathcal{G}$ if and only if $Q \vdash \mathcal{G}$.*

Theorem 4 (Preservation)**.** *If $P \vdash \mathcal{G}$ and $P \longrightarrow Q$, then $Q \vdash \mathcal{G}$.*

Theorem 5 (Progress)**.** *If $R \vdash \mathcal{G}$ then either R is canonical, or there exists R' such that $R \longrightarrow R'$.*

2.4 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 section 2.5.

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

$$\mathsf{tt}_z := z[\textsc{l}].\, z[].\, \mathbf{0} \vdash z : \mathbf{2} \qquad \mathsf{ff}_z := z[\textsc{r}].\, z[].\, \mathbf{0} \vdash z : \mathbf{2}$$

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

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

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

$$\nu xy\,(\mathsf{tt}_x \mid \mathsf{if}(y;\ P;\ Q)) \longrightarrow^* \mathbf{0} \mid P \equiv P \quad \nu xy\,(\mathsf{ff}_x \mid \mathsf{if}(y;\ P;\ Q)) \longrightarrow^* \mathbf{0} \mid Q \equiv P$$

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

$$A := \mathbf{2} \otimes \mathbf{2} \otimes \mathbf{2}^\bot \parr \mathbf{1}$$

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.

$$\begin{aligned}
C_0 &:= x_0[x_e].\, x_0[x_d].\, (\mathsf{ff}_{x_e} \mid \mathsf{tt}_{x_d} \mid x_0 \leftrightarrow r_0) \vdash x_0 : \mathbf{2} \otimes \mathbf{2} \otimes \mathbf{2}^\bot \parr \mathbf{1}, r_0 : \mathbf{2} \otimes \bot \\
C_1 &:= x_1[x_e].\, x_1[x_d].\, (\mathsf{tt}_{x_e} \mid \mathsf{ff}_{x_d} \mid x_1 \leftrightarrow r_1) \vdash x_1 : \mathbf{2} \otimes \mathbf{2} \otimes \mathbf{2}^\bot \parr \mathbf{1}, r_1 : \mathbf{2} \otimes \bot \\
\mathsf{clients} &:= ¿x[x_1].\, ¿x[x_0].\, ¿x[].\, (C_0 \mid C_1) \\
&\quad \vdash x : ¿\left(\mathbf{2} \otimes \mathbf{2} \otimes \mathbf{2}^\bot \parr \mathbf{1}\right), r_0 : \mathbf{2} \otimes \bot, r_1 : \mathbf{2} \otimes \bot
\end{aligned}$$

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 $B := \mathbf{2}$. We initialize the register to false, and forward the final state of the register to u.

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

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:

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

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:

$$\begin{aligned} R_1 &:= \mathsf{if}(y_e;\ \mathsf{if}(y_d;\ S_{111};\ S_{110});\ \mathsf{if}(y_d;\ S_{101};\ S_{100})) \\ &\quad \vdash y_e : \mathbf{2}^\perp, y_d : \mathbf{2}^\perp, y' : \mathbf{2} \otimes \bot, z' : \mathbf{2} \\ R_0 &:= \mathsf{if}(y_e;\ \mathsf{if}(y_d;\ S_{011};\ S_{010});\ \mathsf{if}(y_d;\ S_{001};\ S_{000})) \\ &\quad \vdash y_e : \mathbf{2}^\perp, y_d : \mathbf{2}^\perp, y' : \mathbf{2} \otimes \bot, z' : \mathbf{2} \end{aligned}$$

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:

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

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 $\mathsf{server} := ¡y\{z, z', y'.\, Q\}(i, f).\, P \vdash y : ¡(\mathbf{2}^\perp \parr \mathbf{2}^\perp \parr \mathbf{2} \otimes \bot), u : \mathbf{2}$, and cut:

$$\begin{aligned}
& \nu xy\,(\mathsf{clients} \mid \mathsf{server}) \\
= {} & \nu xy\,(¿x[x_1].\,¿x[x_0].\,¿x[].\,(C_0 \mid C_1) \mid \mathsf{server}) \\
\equiv {} & \nu xy\,(¿x[x_0].\,¿x[x_1].\,¿x[].\,(C_0 \mid C_1) \mid \mathsf{server}) \\
& \hfill (x_0 \text{ preempts } x_1 \text{ using } 2.3.6) \\
\longrightarrow {} & \nu xy\,\nu x_0 y'\,(C_0 \mid ¿x[x_1].\,¿x[].\,C_1 \mid ¡y\{z,z',y'.\,Q\}(z',f).\,(\nu iz\,(P \mid Q))) \\
& \hfill (C_0 \text{ is accepted}) \\
\longrightarrow^* {} & r_0[y_r].\,(\mathsf{tt}_{y_r} \mid r_0().\,\nu xy\,(¿x[x_1].\,¿x[].\,C_1 \mid ¡y\{z,z',y'.\,Q\}(z'',f).\,P')) \\
& \hfill (C_0 \text{ performs CAS}) \\
\longrightarrow {} & r_0[y_r].\,(\mathsf{tt}_{y_r} \mid r_0().\,\nu xy\,\nu x_1 y'\,(C_1 \mid ¿x[].\,\mathbf{0} \mid ¡y\{z,z',y'.\,Q\}(z',f).\,(\nu z''z\,(P' \mid Q)))) \\
& \hfill (C_1 \text{ is accepted}) \\
\longrightarrow^* {} & r_0[y_r].\,(\mathsf{tt}_{y_r} \mid r_0().\,r_1[y_r].\,(\mathsf{tt}_{y_r} \mid r_1().\,\nu xy\,(¿x[].\,\mathbf{0} \mid ¡y\{z,z',y'.\,Q\}(z''',f).\,P''))) \\
& \hfill (C_1 \text{ performs CAS}) \\
\longrightarrow {} & r_0[y_r].\,(\mathsf{tt}_{y_r} \mid r_0().\,r_1[y_r].\,(\mathsf{tt}_{y_r} \mid r_1().\,(\mathbf{0} \mid \nu z'''f\,P''))) \\
& \hfill (\mathsf{server} \text{ starts to finalize}) \\
\longrightarrow^* {} & r_0[y_r].\,(\mathsf{tt}_{y_r} \mid r_0().\,r_1[y_r].\,(\mathsf{tt}_{y_r} \mid r_1().\,\mathsf{ff}_u)) \vdash r_0 : \mathbf{2} \otimes \bot, r_1 : \mathbf{2} \otimes \bot, u : \mathbf{2} \\
& \hfill (\mathsf{server} \text{ finalizes})
\end{aligned}$$

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 section 2.1.1, is modelled.

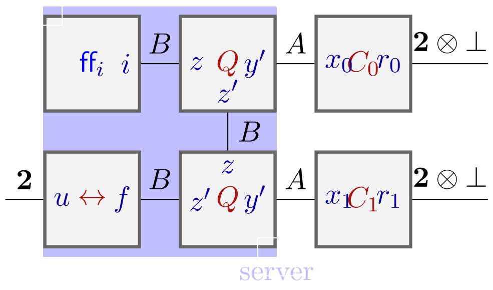

Figure 2.4: Topology of Compare-and-Set protocol, after two server acceptances. Boxes represent processes. Cuts are represented by edges connecting two channels. The dual of each session type is omitted for simplicity.

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.

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. 2.4 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 2.3.6 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.

2.5 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.

2.5.1 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.

$$\begin{array}{rcll}
T, \ldots & ::= & T \multimap T \mid T \to T \mid T + T \mid T \otimes T \mid \mathsf{Unit} \mid T_S & \\
T_S, \ldots & ::= & !T.T_S & \text{(output value of type } T\text{, then behave as } T_S\text{)} \\
& & \mid ?T.T_S & \text{(input value of type } T\text{, then behave as } T_S\text{)} \\
& & \mid T_S \oplus T_S & \text{(select from options)} \\
& & \mid T_S \mathbin{\&} T_S & \text{(offer choice)} \\
& & \mid \mathsf{end}_? \mid \mathsf{end}_! & \text{(end-of-session)} \\
& & \mid \text{¿}T_S & \text{(request } T_S \text{ session)} \\
& & \mid \text{¡}T_S & \text{(serve } T_S \text{ session)}
\end{array}$$

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. [1995, 1999]. The session types are translated as follows:

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

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, $\text{¡}S$ 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.

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

The translation is a homomorphism of involutions:

Lemma 6. $[\![\overline{T_S}]\!] = [\![T_S]\!]^{\perp}$.

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

Definition 3. The set of *unlimited types* is defined inductively as follows.

- $\mathbf{1}$ and $T \to U$ 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.

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

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

Selected typing rules of CSGV are given in fig. 2.5 and fig. 2.6. 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.

$$\left[\!\!\left[\begin{array}{l}\textsc{Recv}\\ \dfrac{\Gamma \vdash M : ?T.T_S}{\Gamma \vdash \mathsf{recv}\ M : T \otimes T_S}\end{array}\right]\!\!\right]_{[} z] := \ [\![M]\!]_z \vdash [\![\Gamma]\!]^\perp, z : [\![T]\!] \otimes [\![T_S]\!]$$

$$\left[\!\!\left[\begin{array}{l}\textsc{Send}\\ \dfrac{\Gamma \vdash M : T \qquad \Delta \vdash N : !T.T_S}{\Gamma, \Delta \vdash \mathsf{send}\ M\ N : T_S}\end{array}\right]\!\!\right]_{[} z] :=$$

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

$$\left[\!\!\left[\begin{array}{l}\textsc{Conn}\\ \dfrac{\Gamma, x : T_S \vdash M : \mathsf{end}_! \qquad \Delta, y : \overline{T_S} \vdash N : T}{\Gamma, \Delta \vdash \mathsf{connect}(x.\,M; y.\,N) : T}\end{array}\right]\!\!\right]_{[} z] :=$$

$$\dfrac{\dfrac{[\![M]\!]_y \vdash [\![\Gamma]\!]^\perp, x : [\![T_S]\!]^\perp, y : \perp \qquad z[].\,\mathbf{0} \vdash z : \mathbf{1}}{\nu yz\,([\![M]\!]_y \mid z[].\,\mathbf{0}) \vdash [\![\Gamma]\!]^\perp, x : [\![T_S]\!]^\perp}\textsc{Cut} \qquad [\![N]\!]_z \vdash [\![\Delta]\!]^\perp, y : [\![T_S]\!], z : [\![T]\!]}{\nu xy\,(\nu yz\,([\![M]\!]_y \mid z[].\,\mathbf{0}) \mid [\![N]\!]_z) \vdash [\![\Gamma]\!]^\perp, [\![\Delta]\!]^\perp, z : [\![T]\!]}\textsc{Cut}$$

Figure 2.5: CSGV Typing Rules and Translation to CSLL: linear session part

$$\left\llbracket \begin{array}{l}\textsc{ReqW}\\ \dfrac{}{x : \text{¿}T_S \vdash \mathsf{eof}_x : \mathsf{end}_!}\end{array} \right\rrbracket_{[} z] := \quad \dfrac{\dfrac{\mathbf{0} \vdash \emptyset}{\text{¿}x[].\,\mathbf{0} \vdash x : \text{¿}\llbracket T_S \rrbracket^{\perp}}}{z().\,\text{¿}x[].\,\mathbf{0} \vdash x : \text{¿}\llbracket T_S \rrbracket^{\perp}, z : \perp}$$

$$\left\llbracket \begin{array}{l}\textsc{ReqA}\\ \dfrac{\Gamma, x' : T_S \vdash M : \mathsf{end}_!}{\Gamma, x : \text{¿}T_S \vdash \mathsf{fork}_x\ x'.\,M : \text{¿}T_S}\end{array} \right\rrbracket_{[} z] :=$$

$$\dfrac{\dfrac{\llbracket M \rrbracket_u \vdash \llbracket \Gamma \rrbracket^{\perp}, x' : \llbracket T_S \rrbracket^{\perp}, u : \perp \qquad v[].\,\mathbf{0} \vdash v : \mathbf{1}}{\nu uv\,(\llbracket M \rrbracket_z \mid v[].\,\mathbf{0}) \vdash \llbracket \Gamma \rrbracket^{\perp}, x' : \llbracket T_S \rrbracket^{\perp}} \quad x \leftrightarrow z \vdash x : \text{¿}\llbracket T_S \rrbracket^{\perp}, z : \text{¡}\llbracket T_S \rrbracket}{\text{¿}x[x'].\,(\nu uv\,(z[].\,\mathbf{0} \mid \llbracket M \rrbracket_u) \mid x \leftrightarrow z) \vdash \llbracket \Gamma \rrbracket^{\perp}, x : \text{¿}\llbracket T_S \rrbracket^{\perp}, z : \text{¡}\llbracket T_S \rrbracket}\ \textsc{HMix2+QueA}$$

$$\left\llbracket \begin{array}{l}\textsc{Serv}\\ \dfrac{\Delta \vdash L : T \qquad z : T, y : T_S \vdash M : T \qquad \Sigma, f : T \vdash N : U}{\Delta, \Sigma, y : \text{¡}T_S \vdash \mathsf{serve}\ y\{L, z.\,M, f.\,N\} : U}\end{array} \right\rrbracket_{[} u] :=$$

$$\dfrac{\dfrac{\llbracket L \rrbracket_i \vdash \llbracket \Delta \rrbracket^{\perp}, i : \llbracket T \rrbracket \qquad \llbracket N \rrbracket_u \vdash \llbracket \Sigma \rrbracket^{\perp}, f : \llbracket T \rrbracket^{\perp}, u : \llbracket U \rrbracket}{\llbracket L \rrbracket_i \mid \llbracket N \rrbracket_u \vdash \llbracket \Delta \rrbracket^{\perp}, i : \llbracket T \rrbracket \mid \llbracket \Sigma \rrbracket^{\perp}, f : \llbracket T \rrbracket^{\perp}, u : \llbracket U \rrbracket} \quad \llbracket M \rrbracket_{z'} \vdash z : \llbracket T \rrbracket^{\perp}, y : \llbracket T_S \rrbracket^{\perp}, z' : \llbracket T \rrbracket}{\text{¡}y\{z, z', y.\,\llbracket M \rrbracket_{z'}\}(i, f).\,(\llbracket L \rrbracket_i \mid \llbracket N \rrbracket_u) \vdash \llbracket \Delta \rrbracket^{\perp}, y : \text{¡}\llbracket T_S \rrbracket^{\perp}, \llbracket \Sigma \rrbracket^{\perp}, u : \llbracket U \rrbracket}\ \textsc{Claro}$$

Figure 2.6: CSGV Typing Rules and Translation to CSLL: shared session part

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 section 2.5.5.

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

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

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

2.5.2 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 $T_S := (?A.\mathsf{end}_?) \,\&\, (!(\mathsf{Unit} + A).\mathsf{end}_?)$. One client receives an A along r_0, and enqueues it. The other one dequeues an element, and sends it along r_1.

$$\begin{aligned}
L &:= \mathsf{empty}\\
M_{enq} &:= \mathsf{let}\ (v, y'') \ = \ \mathsf{recv}\ y'\ \mathsf{in}\\
&\quad\ \mathsf{let}\ \_ \ = \ \mathsf{terminate}\ y''\ \mathsf{in}\ \ \mathsf{enq}(z, v)\\
M_{deq} &:= \mathsf{let}\ (v, z') \ = \ \mathsf{deq}\ z\ \mathsf{in}\\
&\quad\ \mathsf{let}\ \_ \ = \ \mathsf{terminate}\ (\mathsf{send}\ v\ y')\ \mathsf{in}\ \ z'\\
M &:= \mathsf{case}\ y\ \mathsf{of}\ y'.\{M_{enq}, M_{deq}\}
\end{aligned}$$

$$\begin{aligned}
C_0 &:= \mathsf{let}\ (v, r_0') \ = \ \mathsf{recv}\ r_0\ \mathsf{in}\\
&\quad\ \mathsf{let}\ \_ \ = \ \mathsf{terminate}\ r_0'\ \mathsf{in}\\
&\quad\ \mathsf{let}\ x_0' \ = \ \mathsf{send}\ v\ (\mathsf{select}_L\ x_0)\ \mathsf{in}\ \ x_0'\\
C_1 &:= \mathsf{let}\ (v, x_1') \ = \ \mathsf{recv}\ (\mathsf{select}_R\ x_0)\ \mathsf{in}\\
&\quad\ \mathsf{let}\ \_ \ = \ \mathsf{terminate}\ (\mathsf{send}\ v\ r1)\ \mathsf{in}\ \ x_1'\\
\mathsf{clients} &:= \mathsf{let}\ x \ = \ \mathsf{fork}_x\ x_0.\, C_0\ \mathsf{in}\\
&\quad\ \mathsf{let}\ x \ = \ \mathsf{fork}_x\ x_1.\, C_1\ \mathsf{in}\ \ \mathsf{eof}_x
\end{aligned}$$

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

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

2.5.3 Nondeterminism

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

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

(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.

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

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

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

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:

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

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

2.5.4 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:

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

Let

$$\begin{aligned}\mathsf{clients} := {} & \mathsf{let}\ y\ =\ \mathsf{fold}_{¿T_S}\ c\ (\lambda x.\,\lambda v.\,\mathsf{fork}_x\ x'.\,(\mathsf{let}\ v'\ =\ h\ v\ \mathsf{in}\ \ \mathsf{send}\ v'\ x'))\ xs\ \mathsf{in} \\ & \mathsf{eof}_y \\ M := {} & \mathsf{let}\ (v, y')\ =\ \mathsf{recv}\ y\ \mathsf{in}\ \ \mathsf{let}\ \_\ =\ \mathsf{terminate}\ y'\ \mathsf{in}\ \ g\ z\ v\end{aligned}$$

The client protocol is $T_S := !B.\mathsf{end}_!$. 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 $\mathsf{server} := \mathsf{serve'}\ y\{g_0, z.\, M, f.\, f\}$. 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 $\mathsf{serve'}$ to pass unlimited parameters to the server internals.

Putting this system together, we get

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

The fork-join paradigm is often used in industrial parallelization frameworks [Dagum and Menon, 1998, Reinders, 2007, Blumofe et al., 1995, Leijen et al., 2009]. 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].

2.5.5 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 $\overline{T_S}$.

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

The client session type is $C_S := !\mathbb{B}.?\mathbb{B}.\mathsf{end}_!$, and the internal server protocol is $T_S := ?(\mathbb{N} \otimes \mathbb{N}).!\mathbb{B}.\mathsf{end}_?$, 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

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

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

$$
\begin{array}{cc}
w : \overline{T_S} \vdash L : \mathsf{end}_! & f : T_S \vdash N : \mathbf{1} \\
w : \overline{T_S}, z : T_S, y : \overline{C_S} \vdash M : \mathsf{end}_! & y : ¡T_S \vdash \mathsf{server} : \mathbf{1}
\end{array}
$$

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. 2.7. Compared with fig. 2.4 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 $\mathbb{B}$.

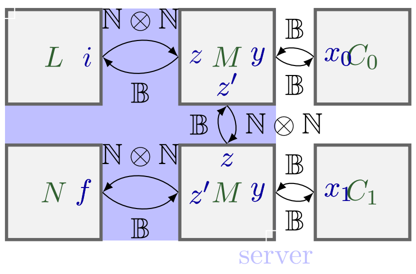

Figure 2.7: Layout of the Keynesian beauty contest, after coexponential reactions but before other reactions. Boxes represents processes whose names are at the center of the boxes. Arrows represents directed mesages between processes with types of the data annotated. Labels on edges of boxes are the names of the channels to the processes.

2.6 Related work

In addition to the general related works as discussed in chapter 1, following are ones more particular to the present extension.

Clients and Servers 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 section 2.1.1.

The present extension centers around the idea that client pool should be represented as a collection of disjoint processes, while the corresponding server internally connected. The same idea was independently developed

[7]This has been confirmed to us by the authors.

by Kokke et al. [2019b]. They 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 section 2.1.1, and does not form a satisfactory model.

Differential Linear Logic The rules for ¿ given in §2.2.2 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.

Appendices

2.A Coexponentials and Logical equivalences

We may derive the following logical equivalences about coexponentials, which are dual to similar laws for the exponentials.

$$¡¡A \equiv ¡A \qquad ¡A \equiv ¡A \parr ¡A \qquad ¡\bot \equiv \bot \qquad ¡(A \mathbin{\&} B) \equiv ¡A \parr ¡B \qquad ¡0 \equiv \bot$$

$$¿¿A \equiv ¿A \qquad ¿A \equiv ¿A \otimes ¿A \qquad ¿1 \equiv 1 \qquad ¿(A \oplus B) \equiv ¿A \otimes ¿B \qquad ¿\top \equiv 1$$

Theorem 7. *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.*

Proof. We first show that the exponential rules are derivable using coexponential rules under the substitution $? \mapsto ¿$. The weakening rule $\dfrac{\vdash \Gamma}{\vdash \Gamma, ?A}\,?w$ is mapped to the derivation $\dfrac{\vdash \Gamma \qquad \dfrac{}{\vdash ¿A}\,¿w}{\vdash \Gamma, ¿A}\,\textsc{Mix}$. The dereliction rule $?d$ is just $¿d$, and the contraction rule $\dfrac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A}\,?c$ is mapped to

$$\dfrac{\vdash \Gamma, ¿A, ¿A \qquad \dfrac{\dfrac{}{\vdash ¡A^{\bot}, ¿A}\,\textsc{Ax} \qquad \dfrac{}{\vdash ¡A^{\bot}, ¿A}\,\textsc{Ax}}{\vdash ¡A^{\bot}, ¡A^{\bot}, ¿A}\,¿c}{\Gamma, ¿A}\,\textsc{BiCut}$$

This leaves promotion. The forklores (iii–v) can be generalised to a bi-implication

$$A_1 \parr \ldots \parr A_n \multimap A_1 \otimes \cdots \otimes A_n \qquad A_1 \otimes \cdots \otimes A_n \multimap A_1 \parr \ldots \parr A_n$$

and hence sequents $\vdash \bigotimes \Delta^\perp, \bigotimes \Delta$ and $\vdash \bigparr \Delta^\perp, \Delta$ for any Δ. With these in hand, we can interpret the promotion rule $\dfrac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A}\,!$ by the derivation

$$\dfrac{\dfrac{}{\vdash ¿\Gamma, \bigparr ¡\Gamma^\perp} \qquad \dfrac{\dfrac{\dfrac{}{\vdash \bigotimes ¿\Gamma, \bigotimes ¡\Gamma^\perp} \qquad \dfrac{\vdash ¿\Gamma, A}{\vdash \bigparr ¿\Gamma, A}\,\bigparr}{\vdash \bigotimes ¿\Gamma, A}\,\textsc{Cut}}{\vdash \bigotimes ¿\Gamma, ¡A}\,¡}{\vdash ¿\Gamma, ¡A}\,\textsc{Cut}$$

In the opposite direction, we show that the coexponentials rules are derivable using exponentials rules under the substitution $¿ \mapsto ?$. As the folklores ensure Mix0 is derivable in this system, we can interpret the weakening rule $\dfrac{}{\vdash ¿A}\,¿w$ by $\dfrac{\dfrac{}{\vdash \cdot}\,\textsc{Mix0}}{\vdash ?A}\,?w$. The dereliction rule $¿d$ is simply $?d$, and the contraction rule $\dfrac{\vdash \Gamma, ¿A \qquad \vdash \Delta, ¿A}{\vdash \Gamma, \Delta, ¿A}\,¿c$ is interpreted by the derivation

$$\dfrac{\dfrac{\vdash \Gamma, ?A \qquad \vdash \Delta, ?A}{\vdash \Gamma, \Delta, ?A, ?A}\,\textsc{Mix}}{\vdash \Gamma, \Delta, ?A}\,?c$$

Finally, the rule $\dfrac{\vdash \otimes ¿\Gamma, A}{\vdash \otimes ¿\Gamma, ¡A}\,¡$ is interpreted in a way similar to promotion, but with the cuts replacing $\otimes$ with $\parr$ happening in the opposite order.

□

2.B Translation of CSGV to CSLL: Omitted rules

Of the functional fragment of CSGV the types are translated to CSLL:

$$\begin{aligned} [\![T \multimap U]\!] &:= [\![T]\!]^\perp \parr [\![U]\!] \\ [\![T \to U]\!] &:= !([\![T]\!]^\perp \parr [\![U]\!]) \end{aligned} \qquad \begin{aligned} [\![T + U]\!] &:= [\![T]\!] \oplus [\![U]\!] \\ [\![T \otimes U]\!] &:= [\![T]\!] \otimes [\![U]\!] \\ [\![\mathsf{Unit}]\!] &:= \mathbf{1} \end{aligned}$$

Omitted rules of CSGV with their translation into CSLL are shown in fig. 2.B.1, fig. 2.B.2 and fig. 2.B.3. Note that some translations use lemma 16.

$$\left\llbracket \begin{array}{l} {\multimap}\, I \\ \dfrac{\Gamma, x : T \vdash L : U}{\Gamma \vdash \lambda x.\, L : T \multimap U} \end{array} \right\rrbracket_{[}\, z] := \dfrac{\llbracket L \rrbracket_z \vdash \llbracket \Gamma \rrbracket^\perp, x : \llbracket T \rrbracket^\perp, z : \llbracket U \rrbracket}{z(x).\, \llbracket L \rrbracket_z \vdash \llbracket \Gamma \rrbracket^\perp, z : \llbracket T \rrbracket^\perp \parr \llbracket U \rrbracket}$$

$$\left\llbracket \begin{array}{l} {\multimap}\, E \\ \dfrac{\Gamma \vdash M : T \multimap U \qquad \Delta \vdash N : T}{\Gamma, \Delta \vdash M\ N : U} \end{array} \right\rrbracket_{[}\, x] :=$$

$$\dfrac{\llbracket M \rrbracket_z \vdash \llbracket \Gamma \rrbracket^\perp, z : \llbracket T \rrbracket^\perp \parr \llbracket U \rrbracket \qquad \dfrac{\llbracket N \rrbracket_{z'} \vdash \llbracket \Delta \rrbracket^\perp, z' : \llbracket T \rrbracket \qquad x \leftrightarrow y \vdash x : \llbracket U \rrbracket, y : \llbracket U \rrbracket^\perp}{y[z'].\, (\llbracket N \rrbracket_{z'} \mid x \leftrightarrow y) \vdash \llbracket \Delta \rrbracket^\perp, x : \llbracket U \rrbracket, y : \llbracket T \rrbracket \otimes \llbracket U \rrbracket^\perp}\otimes}{\nu zy\, (\llbracket M \rrbracket_z \mid y[z'].\, (\llbracket N \rrbracket_{z'} \mid x \leftrightarrow y)) \vdash \llbracket \Gamma \rrbracket^\perp, \llbracket \Delta \rrbracket^\perp, x : \llbracket U \rrbracket}$$

$$\left\llbracket \begin{array}{l} \to I \\ \dfrac{\vec{v} : \vec{V} \vdash L : T \multimap U \qquad \vec{V}\ \text{unlimited}}{\vec{v} : \vec{V} \vdash L : T \to U} \end{array} \right\rrbracket_{[}\, z] :=$$

$$\dfrac{\dfrac{\dfrac{\llbracket L \rrbracket_z \vdash \vec{v} : \llbracket \vec{V} \rrbracket^\perp, z : \llbracket T \rrbracket^\perp \parr \llbracket U \rrbracket}{\vec{v'}[\text{USE}].\, \vec{v}\llbracket L \rrbracket_z \vdash \vec{v'} : ?\llbracket \vec{V} \rrbracket^\perp, z : \llbracket T \rrbracket^\perp \parr \llbracket U \rrbracket}}{!z\{\vec{v'}.\, \vec{v'}[\text{USE}].\, \vec{v}\llbracket L \rrbracket_z\} \vdash \vec{v'} : ?\llbracket \vec{V} \rrbracket^\perp, z : !(\llbracket T \rrbracket^\perp \parr \llbracket U \rrbracket)}}{\overline{?}\vec{v}[\vec{v'}].!z\{\vec{v}.\, \vec{v'}[\text{USE}].\, \vec{v}\llbracket L \rrbracket_z\} \vdash \vec{v} : \llbracket \vec{V} \rrbracket^\perp, z : !(\llbracket T \rrbracket^\perp \parr \llbracket U \rrbracket)}$$

$$\left\llbracket \begin{array}{l} \to E \\ \dfrac{\Gamma \vdash L : T \to U}{\Gamma \vdash L : T \multimap U} \end{array} \right\rrbracket_{[}\, x] := \dfrac{\llbracket L \rrbracket_z \vdash \llbracket \Gamma \rrbracket^\perp, z : !(\llbracket T \rrbracket^\perp \parr \llbracket U \rrbracket) \qquad \dfrac{x \leftrightarrow y \vdash x : \llbracket T \rrbracket^\perp \parr \llbracket U \rrbracket, y : \llbracket T \rrbracket \otimes \llbracket U \rrbracket^\perp}{y'[\text{USE}].\, yx \leftrightarrow y \vdash x : \llbracket T \rrbracket^\perp \parr \llbracket U \rrbracket, y' : ?(\llbracket T \rrbracket \otimes \llbracket U \rrbracket^\perp)}}{\nu zy'\, (\llbracket L \rrbracket_z \mid y'[\text{USE}].\, yx \leftrightarrow y) \vdash \llbracket \Gamma \rrbracket^\perp x : \llbracket T \rrbracket^\perp \parr \llbracket U \rrbracket}$$

$$\left\llbracket \begin{array}{l} w \\ \dfrac{\Gamma \vdash L : T \qquad U\ \text{unlimited}}{\Gamma, x : U \vdash L : T} \end{array} \right\rrbracket_{[}\, z] := \dfrac{\dfrac{\llbracket L \rrbracket_z \vdash \llbracket \Gamma \rrbracket^\perp, z : \llbracket T \rrbracket}{x'[\text{DISP}].\, \llbracket L \rrbracket_z \vdash \llbracket \Gamma \rrbracket^\perp, x' : ?\llbracket U \rrbracket^\perp, z : \llbracket T \rrbracket}}{\overline{?}x[x'].x[\text{DISP}].\, \llbracket L \rrbracket_z \vdash \llbracket \Gamma \rrbracket^\perp, x : \llbracket U \rrbracket^\perp, z : \llbracket T \rrbracket}$$

Figure 2.B.1: Translation from CSGV to CSLL, functional fragments, Part one

$$\left\llbracket^{c} \dfrac{\Gamma, x_0 : U, x_1 : U \vdash L : T \qquad U \text{ unlimited}}{\Gamma, x : U \vdash L[x/x_0][x/x_1] : T} \right\rrbracket_{[} z] :=$$

$$\dfrac{\dfrac{\dfrac{\llbracket L \rrbracket_z \vdash \llbracket \Gamma \rrbracket^{\perp}, x_0 : \llbracket U \rrbracket^{\perp}, x_1 : \llbracket U \rrbracket^{\perp}, z : \llbracket T \rrbracket}{x_0'[\text{USE}].\, x_0 x_1'[\text{USE}].\, x_1 \llbracket L \rrbracket_z \vdash \llbracket \Gamma \rrbracket^{\perp}, x_0' : ?\llbracket U \rrbracket^{\perp}, x_1' : ?\llbracket U \rrbracket^{\perp}, z : \llbracket T \rrbracket}}{x'[\text{DUP}](x_0').\, x_1' x_0'[\text{USE}].\, x_0 x_1'[\text{USE}].\, x_1 \llbracket L \rrbracket_z \vdash \llbracket \Gamma \rrbracket^{\perp}, x' : ?\llbracket U \rrbracket^{\perp}, z : \llbracket T \rrbracket}}{\overline{?}x[x'].x'[\text{DUP}](x_0').\, x_1' x_0'[\text{USE}].\, x_0 x_1'[\text{USE}].\, x_1 \llbracket L \rrbracket_z \vdash \llbracket \Gamma \rrbracket^{\perp}, x : \llbracket U \rrbracket^{\perp}, z : \llbracket T \rrbracket}$$

$$\left\llbracket^{\otimes I} \dfrac{\Gamma \vdash M : T \qquad \Delta \vdash N : U}{\Gamma, \Delta \vdash (M, N) : T \otimes U} \right\rrbracket_{[} z'] :=$$

$$\dfrac{\llbracket M \rrbracket_z \vdash \llbracket \Gamma \rrbracket^{\perp}, z : \llbracket T \rrbracket \qquad \llbracket N \rrbracket_{z'} \vdash \llbracket \Delta \rrbracket^{\perp}, z' : \llbracket U \rrbracket}{z'[z].\, (\llbracket M \rrbracket_z \mid \llbracket N \rrbracket_{z'}) \vdash \llbracket \Gamma \rrbracket^{\perp}, \llbracket \Delta \rrbracket^{\perp}, z' : \llbracket T \rrbracket \otimes \llbracket U \rrbracket}$$

$$\left\llbracket^{\otimes E} \dfrac{\Gamma \vdash M : T \otimes U \qquad \Delta, x : T, y : U \vdash N : V}{\Gamma, \Delta \vdash \mathsf{let}\ (x, y) \ = \ M \ \mathsf{in} \ N : V} \right\rrbracket_{[} z] :=$$

$$\dfrac{\llbracket M \rrbracket_{z'} \vdash \llbracket \Gamma \rrbracket^{\perp}, z' : \llbracket T \rrbracket \otimes \llbracket U \rrbracket \qquad \dfrac{\llbracket N \rrbracket_z \vdash \llbracket \Delta \rrbracket^{\perp}, x : \llbracket T \rrbracket^{\perp}, y : \llbracket U \rrbracket^{\perp}, z : \llbracket V \rrbracket}{y(x).\, \llbracket N \rrbracket_z \vdash \llbracket \Delta \rrbracket^{\perp}, y : \llbracket T \rrbracket^{\perp} \parr \llbracket U \rrbracket^{\perp}, z : \llbracket V \rrbracket}\parr}{\nu z' y\, (\llbracket M \rrbracket_{z'} \mid y(x).\, \llbracket N \rrbracket_z) \vdash \llbracket \Gamma \rrbracket^{\perp}, \llbracket \Delta \rrbracket^{\perp}, z : \llbracket V \rrbracket}$$

$$\left\llbracket^{+I_L} \dfrac{\Gamma \vdash M : T}{\Gamma \vdash \mathsf{inl}\ M : T + U} \right\rrbracket_{[} z] := \quad \dfrac{\llbracket M \rrbracket_z \vdash \llbracket \Gamma \rrbracket^{\perp}, z : \llbracket T \rrbracket}{z[\text{L}].\, \llbracket M \rrbracket_z \vdash \llbracket \Gamma \rrbracket^{\perp}, z : \llbracket T \rrbracket \oplus \llbracket U \rrbracket}$$

$$\left\llbracket^{+E} \dfrac{\Gamma \vdash L : T + U \qquad \Delta, x : T \vdash M : V \qquad \Delta, x : U \vdash N : V}{\Gamma, \Delta \vdash \mathsf{match}\ L\ \mathsf{with}\ x.\{M, N\} : V} \right\rrbracket_{[} z] :=$$

$$\dfrac{\llbracket L \rrbracket_y \vdash \llbracket \Gamma \rrbracket^{\perp}, y : \llbracket T \rrbracket \oplus \llbracket U \rrbracket \qquad \dfrac{\llbracket M \rrbracket_z \vdash \llbracket \Delta \rrbracket^{\perp}, x : \llbracket T \rrbracket^{\perp}, z : \llbracket V \rrbracket \qquad \llbracket N \rrbracket_z \vdash \llbracket \Delta \rrbracket^{\perp}, x : \llbracket U \rrbracket^{\perp}, z : \llbracket V \rrbracket}{x.\mathsf{case}\{\text{L}:\llbracket M \rrbracket_z, \text{R}:\llbracket N \rrbracket_z\} \vdash \llbracket \Delta \rrbracket^{\perp}, x : \llbracket T \rrbracket^{\perp} \mathbin{\&} \llbracket U \rrbracket^{\perp}, z : V}\mathbin{\&}}{\nu xy\, (\llbracket L \rrbracket_y \mid x.\mathsf{case}\{\text{L}:\llbracket M \rrbracket_z, \text{R}:\llbracket N \rrbracket_z\}) \vdash \llbracket \Gamma \rrbracket^{\perp}, \llbracket \Delta \rrbracket^{\perp}, z : \llbracket V \rrbracket}$$

Figure 2.B.2: Translation from CSGV to CSLL, functional fragments, Part two

$$\left\llbracket \begin{array}{l}\textsc{SelectL}\\ \dfrac{\Gamma \vdash M : T_S \oplus U_S}{\Gamma \vdash \mathsf{select}_L\ M : T_S}\end{array} \right\rrbracket_{[} y] :=$$

$$\dfrac{\llbracket M \rrbracket_z \vdash \llbracket\Gamma\rrbracket^\perp, z : \llbracket T_S\rrbracket \mathbin{\&} \llbracket U_S\rrbracket \qquad \dfrac{x \leftrightarrow y \vdash x : \llbracket T_S\rrbracket^\perp, y : \llbracket T_S\rrbracket}{x[\mathrm{L}].\, x \leftrightarrow y \vdash x : \llbracket T_S\rrbracket^\perp \oplus \llbracket U_S\rrbracket^\perp, y : \llbracket T_S\rrbracket}\ \oplus}{\nu xz\,(\llbracket M\rrbracket_z \mid x[\mathrm{L}].\, x\leftrightarrow y) \vdash \llbracket\Gamma\rrbracket^\perp, y:\llbracket T_S\rrbracket}$$

$$\left\llbracket \begin{array}{l}\textsc{Case}\\ \dfrac{\Gamma \vdash L : T_S \mathbin{\&} U_S \qquad \Delta, x : T_S \vdash M : V \qquad \Delta, x : U_S \vdash N : V}{\Gamma, \Delta \vdash \mathsf{case}\ L\ \mathsf{of}\ x.\{M, N\} : V}\end{array} \right\rrbracket_{[} z] :=$$

$$\dfrac{\llbracket L\rrbracket_y \vdash \llbracket\Gamma\rrbracket^\perp, y : \llbracket T_S\rrbracket \oplus \llbracket U_S\rrbracket \qquad \dfrac{\llbracket M\rrbracket_z \vdash \llbracket\Delta\rrbracket^\perp, x : \llbracket T_S\rrbracket^\perp, z : \llbracket V\rrbracket \qquad \llbracket N\rrbracket_z \vdash \llbracket\Delta\rrbracket^\perp, x : \llbracket U_S\rrbracket^\perp, z : \llbracket V\rrbracket}{x.\mathsf{case}\{\mathrm{L}{:}\llbracket M\rrbracket_z, \mathrm{R}{:}\llbracket N\rrbracket_z\} \vdash \llbracket\Delta\rrbracket^\perp, x : \llbracket T_S\rrbracket^\perp \mathbin{\&} \llbracket U_S\rrbracket^\perp, z : \llbracket V\rrbracket}\ \&}{\nu xy\,(\llbracket L\rrbracket_y \mid x.\mathsf{case}\{\mathrm{L}{:}\llbracket M\rrbracket_z, \mathrm{R}{:}\llbracket N\rrbracket_z\}) \vdash \llbracket\Gamma\rrbracket^\perp, \llbracket\Delta\rrbracket^\perp, z : \llbracket V\rrbracket}\ \textsc{Cut}$$

Figure 2.B.3: Translation from CSGV to CSLL, omitted rules of linear fragments

2.C CSLL: Metatheoretic Proofs

Proof of lemma 3. By induction on $P \equiv Q$. We prove one direction, the other one being entirely analogous. Moreover, the congruence cases are trivial. $P \mid \mathbf{0} \equiv P$, commutativity, and associativity follow from the structure of hyperenvironments. Link-commutativity follows from the involutive property of $(-)^\perp$.

Case(Res-Par).

Then $P = \nu xy\,(R \mid S)$ and $Q = R \mid \nu xy\, S$ where $x, y \notin \textsc{Fn}(R)$. We must then have that $R \vdash \mathcal{H}$ where $x, y \notin \mathcal{H}$ (using lemma 12) and $S \vdash \mathcal{I} \mid \Gamma, x : A \mid \Delta, y : A^\perp$, where $\mathcal{G} = \mathcal{H} \mid \mathcal{I} \mid \Gamma, \Delta$. Hence, we can derive that $Q = R \mid \nu xy\, S \vdash \mathcal{G}$.

Case(Res-Res).

Then $P = \nu xy\, \nu zw\, R$ and $Q = \nu zw\, \nu xy\, R$ for some R. We must invert $P \vdash \mathcal{G}$. This generates many cases: for example, it could be that $R \vdash \mathcal{G}' \mid \Gamma, x : A, z : B \mid \Delta, y : A^\perp \mid \Sigma, w : B^\perp$ where $\mathcal{G} = \mathcal{G}' \mid \Gamma, \Delta, \Sigma$, whence $Q = \nu zw\, \nu xy\, R \vdash \mathcal{G}$. The other cases are similar.

Case(Res-Pre). We show the case for $\nu xy\, z[w].\, P \equiv z[w].\, \nu xy\, P$, with that of other prefixes being similar. We have

$$\dfrac{\dfrac{P \vdash \mathcal{G} \mid \Sigma, y : C^\perp \mid \Gamma, z : A, x : C \mid \Delta, w : B}{z[w].\, P \vdash \mathcal{G} \mid \Sigma, y : C^\perp \mid \Gamma, z : B \otimes A, x : C, \Delta}}{\nu xy\, z[w].\, P \vdash \mathcal{G} \mid \Sigma, \Gamma, z : B \otimes A, \Delta}$$

and therefore

$$\dfrac{\dfrac{P \vdash \mathcal{G} \mid \Sigma, y : C^\perp \mid \Gamma, z : A, x : C \mid \Delta, w : B}{\nu xy\, P \vdash \mathcal{G} \mid \Sigma, \Gamma, z : A \mid \Delta, w : B}}{z[w].\, \nu xy\, P \vdash \mathcal{G} \mid \Sigma, \Gamma, z : B \otimes A, \Delta}$$

the other case has x, y, z, w in separate environments and are simpler.

CASE(PRE-PAR). We show the case for $x[y].\, (P \mid Q) \equiv P \mid x[y].\, Q$, with that of other prefixes being similar. We have

$$\dfrac{\dfrac{P \vdash \mathcal{G} \qquad Q \vdash \mathcal{H} \mid \Gamma, x : A \mid \Delta, y : B}{P \mid Q \vdash \mathcal{G} \mid \mathcal{H} \mid \Gamma, x : A \mid \Delta, y : B}}{x[y].\, (P \mid Q) \vdash \mathcal{G} \mid \mathcal{H} \mid \Gamma, \Delta, x : B \otimes A}$$

and therefore

$$\dfrac{P \vdash \mathcal{G} \qquad \dfrac{Q \vdash \mathcal{H} \mid \Gamma, x : A \mid \Delta, y : B}{x[y].\, Q \vdash \mathcal{H} \mid \Gamma, \Delta, x : B \otimes A}}{P \mid x[y].\, Q \vdash \mathcal{G} \mid \mathcal{H} \mid \Gamma, \Delta, x : B \otimes A}$$

□

Lemma 8. *If $P \equiv Q$, then P is canonical if and only if Q is canonical.*

Proof. Straightforward by induction on $P \equiv Q$. □

Lemma 9 (Separation)**.** *If $T \vdash \Gamma_0 \mid \cdots \mid \Gamma_{n-1}$, then there exist $T_i \vdash \Gamma_i$ for $0 \le i < n$ such that $T \equiv T_0 \mid \cdots \mid T_{n-1}$. Moreover, if T is canonical, then every T_i is canonical.*

Proof. We prove the first claim by induction on $T \vdash \Gamma_0 \mid \cdots \mid \Gamma_{n-1}$. We show only the following cases; all other cases are either trivial or similar.

CASE(HMIX2). Then $T = P \mid Q$, and after appropriately reordering the hyperenvironment we have $P \vdash \Gamma_0 \mid \cdots \mid \Gamma_{m-1}$ and $Q \vdash \Gamma_m \mid \cdots \mid \Gamma_{n-1}$ with $m \le n$. By the IH we have $T_i \vdash \Gamma_i$ for $0 \le i < n$, with $P \equiv T_0 \mid \cdots \mid T_{m-1}$, and $Q \equiv T_m \mid \cdots \mid T_{n-1}$. We then have $P \mid Q \equiv T_0 \mid \cdots \mid T_{n-1} \equiv T$, as $\equiv$ is a congruence.

Case(Cut). Then $T = \nu xy\, P$, and after appropriately reordering the hyperenvironment we have

$$P \vdash \Gamma_0 \mid \cdots \mid \Gamma_{n-2} \mid \Delta_0, x : A \mid \Delta_1, y : A^{\perp}$$

where $\Gamma_{n-1} = \Delta_0, \Delta_1$. By the IH we have $P_i \vdash \Gamma_i$ for $0 \leq i < n-1$, $P_{n-1} \vdash \Delta_0, x : A$, and $P_n \vdash \Delta_1, y : A^{\perp}$, with $P \equiv P_0 \mid \cdots \mid P_n$. The result follows, as $\nu xy\, (P_{n-1} \mid P_n) \vdash \Gamma_{n-1}$. and by (Res-Par)

$$\nu xy\, P \equiv \nu xy\, (P_0 \mid \cdots \mid P_{n-1} \mid P_n) \equiv P_0 \mid \cdots \mid \nu xy\, (P_{n-1} \mid P_n)$$

Case(Tensor). Then $T = x[y].\, P$, and after appropriately reordering the hyperenvironment we have

$$P \vdash \Gamma_0 \mid \ldots \Gamma_{n-2} \mid \Delta_0, x : A \mid \Delta_1, y : B$$

where $\Gamma_{n-1} = \Delta_0, \Delta_1$. By the IH we have $P_i \vdash \Gamma_i$ for $0 \leq i < n-1$, $P_{n-1} \vdash \Delta_0, x : A$ and $P_n \vdash \Delta_1, y : B$ with $P \equiv P_0 \mid \cdots \mid P_n$. The result follows, as $x[y].\, (P_{n-1} \mid P_n) \vdash \Gamma_{n-1}, x : B \otimes A$, and by ()

The second claim follows by lemma 8, and the fact subterms of canonical terms are canonical. □

Lemma 10 (Local Progress)**.** *If $P \vdash \Gamma, x : A$ and $Q \vdash \Delta, y : A^{\perp}$ and both P and Q are canonical, then there exists an R such that $\nu xy\, (P \mid Q) \longrightarrow R$.*

Proof. By induction on P and Q. Note the two are symmetric which we will exploit to omit some cases. The type judgment implies neither P nor Q can be $\mathbf{0}$. They cannot be of the form $\nu xy\, S$ either, for they would not be canonical.

- If $P = P_0 \mid P_1$, then it must be that $P_1 = \mathbf{0}$ without loss of generality. We have that $P_0 \mid \mathbf{0} \equiv P_0$ by Par-Unit. Apply induction hypothesis on P_0 we get $\nu xy\, (P_0 \mid Q) \longrightarrow R$. Use Eq we have $\nu xy\, (P \mid Q) \longrightarrow R$. Similar when $Q = Q_0 \mid Q_1$.

- If $P = a \leftrightarrow b$, it must be that $x = b$, so we can reduce by Link. Similar for Q.

- The remaining scenarios are where $P = \pi_z.\, P'$, or $P = z.\mathsf{case}\{\text{L}{:}P_0, \text{R}{:}P_1\}$ or $P = !z\{\vec{z'}.\, P'\}$, and similarly $Q = \pi_w.\, Q'$, or $Q = w.\mathsf{case}\{\text{L}{:}Q_0, \text{R}{:}Q_1\}$ or $Q = !w\{\vec{w'}.\, Q'\}$. If $z = x$ and $w = y$, then one of the reaction axioms apply; otherwise we can assume WLOG that $z \neq x$ (and of course $z \neq y$), and take cases of P.

  – If $P = \pi_z.\, P'$, we have that $\nu xy\, (\pi_z.\, P' \mid Q) \equiv \nu xy\, \pi_z.\, (P' \mid Q) \equiv \pi_z.\, \nu xy\, (P' \mid Q)$, given by Pre-Par and Res-Pre accordingly. Note that in the second equivalence, π_z and νxy do not cross $\textsc{Fn}(P' \mid Q)$ because the left hand side is well-typed. By induction hypothesis we have $\nu xy\,(P' \mid Q) \longrightarrow R$, we therefore have $\nu xy\,(P \mid Q) \longrightarrow \pi_z.\,R$ by Pre and Eq.

– If $P = z.\mathsf{case}\{\textsc{l}{:}P_0, \textsc{r}{:}P_1\}$, the commuting conversion Case-Comm applies.

– If $P = !z\{\vec{z'}.\,P'\}$ where $x \in \vec{z'}$. We know $x : A = ?B$ for some B and thus $y : A^\perp = !B^\perp$. We check if $w = y$. If so, we have $Q = !w\{\vec{w'}.\,Q'\}$ and thus OfCourse-Comm applies; otherwise, we know that Q cannot be of the form $!w\{\vec{w'}.\,Q'\}$ where $y \in \vec{w'}$ (because $y : !B^\perp$ breaks OfCourse requirement that $\vec{w'} : ?\vec{B}$), which means $Q = \pi_w Q'$ or $Q = w.\mathsf{case}\{\textsc{l}{:}Q_0, \textsc{r}{:}Q_1\}$ and can be handled similarly as the previous two cases.

□

Theorem 11 (Progress). *If $R \vdash \mathcal{G}$ then either R is canonical, or there exists R' such that $R \longrightarrow R'$.*

Proof. By induction on $R \vdash \mathcal{G}$.

- If $R = \mathbf{0}$ or $x \leftrightarrow y$, then it is canonical.
- Suppose $R = P \mid Q$. If both P and Q are canonical, then so is R. Otherwise, if P is not canonical, then by the IH we have $P \longrightarrow P'$ for some R', and thus $P \mid Q \longrightarrow P' \mid Q$ by ParL. Similarly for Q.
- Suppose $R = \pi_y.\,P$. If P is canonical then so is R. Otherwise P is not canonical, and by the IH $P \longrightarrow P'$, and thus $\pi_y.\,P \longrightarrow \pi_y.\,P'$ for some P' by Pre.
- Suppose $R = y.\mathsf{case}\{\textsc{l}{:}P, \textsc{r}{:}Q\}$ or $R = !y\{\vec{w}.\,P\}$, then it is canonical.
- Suppose $R = \nu xy\,P$, with $P \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : A^\perp$. If P is not canonical then by the IH we have $P \longrightarrow P'$ for some P', and thus $\nu xy\,P \longrightarrow \nu xy\,P'$ by Res. If P is canonical, by lemma 9 we have that $P \equiv P_0 \mid \cdots \mid P_n \mid P_{n+1}$ where $P_n \vdash \Gamma, x : A$ and $P_{n+1} \vdash \Delta, y : A^\perp$. Note that both P_n and P_{n+1} are canonical. Hence we have $R \equiv \nu xy\,(P_0 \mid \cdots \mid P_{n+1})$. By Res-Par and lemma 12 we obtain $R \equiv P_0 \mid \cdots \mid P_{n-1} \mid \nu xy\,(P_n \mid P_{n+1})$. Local progress (lemma 10) yields $\nu xy\,(P_n \mid P_{n+1}) \longrightarrow R'$, which gives $R \longrightarrow P_0 \mid \cdots \mid P_{n-1} \mid R'$ by ParL.

□

Lemma 12. *If $P \vdash \mathcal{G}$, then $\textsc{Fn}(P) = \textsc{Fn}(\mathcal{G})$.*

Proof. Straightforward by induction on $P \vdash \mathcal{G}$. $\square$

Lemma 13. *If* $P \vdash \mathcal{G} \mid \Gamma, y : A$ *and* $x \notin \mathcal{G}, \Gamma$*, then* $P[x/y] \vdash \mathcal{G} \mid \Gamma, x : A$*.*

Proof. Straightforward by induction on $P \vdash \mathcal{G} \mid \Gamma, y : A$. $\square$

Theorem 14 (Preservation)**.** *If* $P \vdash \mathcal{G}$ *and* $P \longrightarrow Q$*, then* $Q \vdash \mathcal{G}$*.*

Proof. By induction on $P \longrightarrow Q$. We show the nontrivial cases of top-level cuts, and the commuting conversions.

Case(Eq). Suppose $P \equiv P' \longrightarrow Q' \equiv Q$. Then the result follows by the IH and two applications of lemma 3.

Case(Case-Comm). The redex is $\nu xy\,(z.\mathsf{case}\{\textsc{l}{:}P_0, \textsc{r}{:}P_1\} \mid Q)$ and typed.

$$\dfrac{\dfrac{P_0 \vdash \Gamma, x : C, z : A \qquad P_1 \vdash \Gamma, x : C, z : B}{z.\mathsf{case}\{\textsc{l}{:}P_0, \textsc{r}{:}P_1\} \vdash \Gamma, x : C, z : A \mathbin{\&} B} \qquad Q \vdash \Delta, y : C^{\perp}}{\nu xy\,(z.\mathsf{case}\{\textsc{l}{:}P_0, \textsc{r}{:}P_1\} \mid Q) \vdash \Gamma, \Delta, z : A \mathbin{\&} B}$$

and therefore

$$\dfrac{\nu xy\,(P_0 \mid Q) \vdash \Gamma, \Delta, z : A \qquad \nu xy\,(P_1 \mid Q) \vdash \Gamma, \Delta, z : B}{z.\mathsf{case}\{\textsc{l}{:}\nu xy\,(P_0 \mid Q), \textsc{r}{:}\nu xy\,(P_1 \mid Q)\} \vdash \Gamma, \Delta, z : A \mathbin{\&} B}$$

Case(OfCourse-Comm). The redex is $\nu xy\,(!z\{x\vec{w}.\,P\} \mid !y\{\vec{v}.\,Q\})$ and typed.

$$\dfrac{\dfrac{P \vdash \vec{w} : ?\vec{B}, z : A, x : ?C}{!z\{x\vec{w}.\,P\} \vdash \vec{w} : ?\vec{B}, z : !A, x : ?C} \qquad \dfrac{Q \vdash \vec{v} : ?\vec{D}, y : !C^{\perp}}{!y\{\vec{v}.\,Q\} \vdash \vec{v} : ?\vec{D}, y : !C^{\perp}}}{\nu xy\,(!z\{x\vec{w}.\,P\} \mid !y\{\vec{v}.\,Q\}) \vdash \vec{w} : ?\vec{B}, z : ?A, \vec{v} : ?\vec{D}}$$

and therefore

$$\dfrac{\dfrac{P \vdash \vec{w} : ?\vec{B}, z : A, x : ?C \qquad !y\{\vec{v}.\,Q\} \vdash \vec{v} : ?\vec{D}, y : !C^{\perp}}{\nu xy\,(P \mid !y\{\vec{v}.\,Q\}) \vdash \vec{w} : ?\vec{B}, z : A, \vec{v} : ?\vec{D}}}{!z\{\vec{v}\vec{w}.\,\nu xy\,(P \mid !y\{\vec{v}.\,Q\})\} \vdash \vec{w} : ?\vec{B}, z : !A, \vec{v} : ?\vec{D}}$$

Case(Link). Then the redex is $\nu xy\,(z \leftrightarrow x \mid P)$ and the last steps of the typing derivation must have been

$$\dfrac{\dfrac{z \leftrightarrow x \vdash z : A^{\perp}, x : A \qquad P \vdash \mathcal{G} \mid \Gamma, y : A^{\perp}}{z \leftrightarrow x \mid P \vdash z : A^{\perp}, x : A \mid \mathcal{G} \mid \Gamma, y : A^{\perp}}}{\nu xy\,(z \leftrightarrow x \mid P) \vdash \mathcal{G} \mid \Gamma, z : A^{\perp}}$$

and therefore $P[z/y] \vdash \mathcal{G} \mid \Gamma, z : A^{\perp}$ by lemma 13 because $z \notin \textsc{Fn}(P)$. (otherwise the redex would not be well-typed)

Case(One-Bot). Then the redex is $\nu xy\,(x().\,P \mid y[].\,Q)$ and the last steps of the typing derivation must have been

$$\dfrac{\dfrac{P \vdash \mathcal{G} \mid \Gamma}{x().\,P \vdash \mathcal{G} \mid \Gamma, x : \bot} \qquad \dfrac{Q \vdash \mathcal{H}}{y[].\,Q \vdash \mathcal{H} \mid y : \mathbf{1}}}{\nu xy\,(x().\,P \mid y[].\,Q) \vdash \mathcal{G} \mid \mathcal{H} \mid \Gamma}$$

Hence, we have

$$\dfrac{P \vdash \mathcal{G} \mid \Gamma \qquad Q \vdash \mathcal{H}}{P \mid Q \vdash \mathcal{G} \mid \mathcal{H} \mid \Gamma}$$

Case(Tensor-Par). Then the redex is $\nu xy\, x[z].\,P \mid y(w).\,Q$, and the last steps of the typing derivation must have been

$$\dfrac{\dfrac{P \vdash \mathcal{G} \mid \Gamma, z : A \mid \Delta, x : B}{x[z].\,P \vdash \mathcal{G} \mid \Gamma, \Delta, x : A \otimes B} \qquad \dfrac{Q \vdash \mathcal{H} \mid \Sigma, w : A^\perp, y : B^\perp}{y(w).\,Q \vdash \mathcal{H} \mid \Sigma, y : A^\perp \mathbin{⅋} B^\perp}}{\nu xy\,(x[z].\,P \mid y(w).\,Q) \vdash \mathcal{G} \mid \mathcal{H} \mid \Gamma, \Delta, \Sigma}$$

so that $\mathcal{G} = \Gamma, \Delta, \Sigma$. Therefore, we can infer that

$$\dfrac{P \vdash \mathcal{G} \mid \Gamma, z : A \mid \Delta, x : B \qquad Q \vdash \mathcal{H} \mid \Sigma, w : A^\perp, y : B^\perp}{\nu xy\,\nu zw\,(P \mid Q) \vdash \mathcal{G} \mid \mathcal{H} \mid \Gamma, \Delta, \Sigma}$$

Case(PlusL-With). Then the redex is $\nu xy\,(x[\text{L}].\,P \mid y.\mathsf{case}\{\text{L}{:}Q_l, \text{R}{:}Q_r\})$, and the last steps of the typing derivation must have been

$$\dfrac{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A}{x[\text{L}].\,P \vdash \mathcal{G} \mid \Gamma, x : A \oplus B} \qquad \dfrac{Q_l \vdash \Delta, y : A^\perp \qquad Q_r \vdash \Delta, y : B^\perp}{y.\mathsf{case}\{\text{L}{:}Q_l, \text{R}{:}Q_r\} \vdash \Delta, y : A^\perp \mathbin{\&} B^\perp}}{\nu xy\,(x[\text{L}].\,P \mid y.\mathsf{case}\{\text{L}{:}Q_l, \text{R}{:}Q_r\}) \vdash \mathcal{G} \mid \Gamma, \Delta}$$

Hence,

$$\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A \qquad Q_l \vdash \Delta, y : A^\perp}{\nu xy\,(P \mid Q_l) \vdash \mathcal{G} \mid \Gamma, \Delta}$$

Case(Claro-QueW). This is the case of an empty client pool. The redex must be

$$\nu xy\,(\text{¿}x[].\,S \mid \text{¡}y\{z, z', y'.\,Q\}(i, f).\,P)$$

and the last steps in the typing derivation must have been

$$\dfrac{\mathcal{D} \vdash \mathcal{G} \mid x : \text{¿}A \qquad \dfrac{P \vdash \mathcal{H} \mid \Delta, i : B \mid \Sigma, f : B^\perp \qquad Q \vdash z : B^\perp, z' : B, y' : A^\perp}{\text{¡}y\{z, z', y'.\,Q\}(i, f).\,P \vdash \mathcal{H} \mid \Delta, y : \text{¡}A^\perp, \Sigma}\ \textsc{Claro}}{\nu xy\,(\mathcal{D} \mid \text{¡}y\{z, z', y'.\,Q\}(i, f).\,P) \vdash \mathcal{G} \mid \mathcal{H} \mid \Delta, \Sigma}\ \textsc{Cut}$$

where $\mathcal{D} := \dfrac{S \vdash \mathcal{G}}{\text{¿}x[].\,S \vdash \mathcal{G} \mid x : \text{¡}A}$. Hence,

$$\dfrac{S \vdash \mathcal{G} \qquad \dfrac{P \vdash \mathcal{H} \mid \Delta, i : B \mid \Sigma, f : B^\perp}{\nu i f\, P \vdash \mathcal{H} \mid \Delta, \Sigma}}{S \mid \nu i f\, P \vdash \mathcal{G} \mid \mathcal{H} \mid \Delta, \Sigma}$$

Case(Claro-QueA). Then the redex is

$$\nu xy\,(\text{¿}x[x'].\,S \mid \text{¡}y\{z, z', y'.\,Q\}(i, f).\,P)$$

The last few steps in the typing derivation must have been

$$\dfrac{\begin{array}{c}\mathcal{D} \vdash \mathcal{G} \mid \Gamma, \Gamma', x : \text{¿}A \\ \dfrac{P \vdash \mathcal{H} \mid \Delta, i : B \mid \Sigma, f : B^\perp \qquad Q \vdash z : B^\perp, z' : B, y' : A^\perp}{\text{¡}y\{z, z', y'.\,Q\}(i, f).\,P \vdash \mathcal{H} \mid \Delta, y : \text{¡}A^\perp, \Sigma}\,\textsc{Claro}\end{array}}{\nu xy\,(\mathcal{D} \mid \text{¡}y\{z, z', y'.\,Q\}(i, f).\,P) \vdash \mathcal{G} \mid \mathcal{H} \mid \Gamma, \Gamma', \Delta, \Sigma}\,\textsc{Cut}$$

where

$$\mathcal{D} := \dfrac{S \vdash \mathcal{G} \mid \Gamma, x : \text{¿}A \mid \Gamma', x' : A}{\text{¿}x[x'].\,S \vdash \mathcal{G} \mid \Gamma, \Gamma', x : \text{¿}A}$$

Therefore,

$$\dfrac{S \vdash \mathcal{G} \mid \Gamma, x : \text{¿}A \mid \Gamma', x' : A \qquad \mathcal{D} \vdash \mathcal{H} \mid \Delta, y' : A, \Sigma, y : \text{¡}A^\perp}{\nu xy\,\nu x'y'\,(S \mid \mathcal{D}) \vdash \mathcal{G} \mid \mathcal{H} \mid \Gamma, \Gamma', \Delta, \Sigma}$$

where $\mathcal{D}$ is

$$\dfrac{\begin{array}{c}\nu iz\,(P \mid Q) \vdash \mathcal{H} \mid \Delta, z' : B, y' : A^\perp \mid \Sigma, f : B^\perp \\ Q \vdash z : B^\perp, z' : B, y' : A^\perp\end{array}}{\text{¡}y\{z, z', y'.\,Q\}(z', f).\,(\nu iz\,(P \mid Q)) \vdash \mathcal{H} \mid \Delta, y' : A^\perp, \Sigma, y : \text{¡}A^\perp}$$

Case(OfCource-WhyNotW).

$$\dfrac{\dfrac{P \vdash \mathcal{G} \mid \Gamma}{x[\textsc{disp}].\,P \vdash \mathcal{G} \mid \Gamma, x : ?A} \qquad \dfrac{Q \vdash \vec{z} : ?\vec{B}, y : A^\perp}{!y\{\vec{z}.\,Q\} \vdash \vec{z} : ?\vec{B}, y : !A^\perp}}{\nu xy\,(x[\textsc{disp}].\,P \mid !y\{\vec{z}.\,Q\}) \vdash \mathcal{G} \mid \Gamma, \vec{z} : ?\vec{B}}$$

and therefore

$$\dfrac{P \vdash \mathcal{G} \mid \Gamma}{\vec{z}[\textsc{disp}].\,P \vdash \mathcal{G} \mid \Gamma, \vec{z} : ?\vec{B}}$$

CASE(OFCOURSE-WHYNOTD).

$$\dfrac{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x' : A}{x[\mathrm{USE}].\, x'P \vdash \mathcal{G} \mid \Gamma, x : ?A} \qquad \dfrac{Q \vdash \vec{z} : ?\vec{B}, y : A^{\perp}}{!y\{\vec{z}.\, Q\} \vdash \vec{z} : ?\vec{B}, y : ¡A^{\perp}}}{\nu xy\,(x[\mathrm{USE}].\, x'P \mid !y\{\vec{z}.\, Q\}) \vdash \mathcal{G} \mid \Gamma, \vec{z} : ?\vec{B}}$$

and therefore

$$\dfrac{P \vdash \mathcal{G} \mid \Gamma, x' : A \qquad Q \vdash \vec{z} : ?\vec{B}, y : A^{\perp}}{\nu x'y\,(P \mid Q) \vdash \mathcal{G} \mid \Gamma, \vec{z} : ?\vec{B}}$$

CASE(OFCOURSE-WHYNOTC).

$$\dfrac{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x_0 : ?A, x_1 : ?A}{x[\mathrm{DUP}](x_0).\, x_1P \vdash \mathcal{G} \mid \Gamma, x : ?A} \qquad \dfrac{Q \vdash \vec{z} : ?\vec{B}, y : A^{\perp}}{!y\{\vec{z}.\, Q\} \vdash \vec{z} : ?\vec{B}, y : !A^{\perp}}}{\nu xy\,(x[\mathrm{DUP}](x_0).\, x_1P \mid !y\{\vec{z}.\, Q\}) \vdash \mathcal{G} \mid \Gamma, \vec{z} : ?\vec{B}}$$

and therefore

$$\dfrac{\mathcal{D} \vdash \mathcal{G} \mid \Gamma, \vec{z_0} : ?\vec{B}, \vec{z_1} : ?\vec{B}}{\vec{z}[\mathrm{DUP}](\vec{z_0}).\, \vec{z_1}\mathcal{D} \vdash \mathcal{G} \mid \Gamma, \vec{z} : ?\vec{B}}$$

where $\mathcal{D}$ is

$$\dfrac{P \vdash \mathcal{G} \mid \Gamma, x_0 : ?A, x_1 : ?A \qquad \dfrac{Q[z_0/z][y_0/y] \vdash \vec{z_0} : ?\vec{B}, y_0 : A^{\perp}}{!y_0\{\vec{z_0}.\, Q[z_0/z][y_0/y]\} \vdash \vec{z_0} : ?\vec{B}, y_0 : !A^{\perp}} \qquad \dfrac{Q[z_1/z][y_1/y] \vdash \vec{z_1} : ?\vec{B}, y_1 : A^{\perp}}{!y_1\{\vec{z_1}.\, Q[z_1/z][y_1/y]\} \vdash \vec{z_1} : ?\vec{B}, y_1 : !A^{\perp}}}{\begin{array}{c}\nu x_0y_0\, \nu x_1y_1\,(P \mid !y_1\{\vec{z_1}.\, Q[z_1/z][y_1/y]\} \mid !y_0\{\vec{z_0}.\, Q[z_0/z][y_0/y]\}) \\ \vdash \mathcal{G} \mid \Gamma, \vec{z_0} : ?\vec{B}, z_1 : ?\vec{B}\end{array}}$$

□

Lemma 15. $[\![\overline{T_S}]\!] = [\![T_S]\!]^{\perp}$.

Proof. By simple induction. □

Lemma 16. *If T is unlimited, we have the following derivable rule in CSLL.*

$$\textit{POSITIVE}\ \dfrac{P \vdash \mathcal{G} \mid \Gamma, x' : ?[\![T]\!]^{\perp}}{\overline{?}x[x'].P \vdash \mathcal{G} \mid \Gamma, x : [\![T]\!]^{\perp}}$$

The above lemma means that all unlimited types enjoy contraction and weakening. Some session types, such as $end_?$, also enjoy such properties: see e.g. [Gay and Vasconcelos, 2010, §5]. However, in order to retain the good properties of termination and deadlock-freedom, we insist that all channels are used linearly, and carefully closed at the end.

Proof. It is given by the well-known fact that $!A$, $\mathbf{1}$ and $\mathbf{0}$ are always positive, that $\otimes, \oplus$ preserve positivity, and that our system (without server and client) is equivalent to linear logic in terms of expressivity [Kokke et al., 2019a, §2.3]. More concretely, we derive the rule by induction on T.

- If T is Unit we have

$$\dfrac{\dfrac{y[].\,\mathbf{0} \vdash y : \mathbf{1}}{!y\{.\,y[].\,\mathbf{0}\} \vdash y : !\mathbf{1}}}{x().\,!y\{.\,y[].\,\mathbf{0}\} \vdash y : !\mathbf{1}, x : \bot}$$

  Cut y of this with x' of P and we are done.

- If T is $U \to V$, we have

$$\dfrac{y \leftrightarrow x \vdash y : !(\llbracket U \rrbracket \otimes \llbracket V \rrbracket^\perp), x : ?(\llbracket U \rrbracket^\perp \parr \llbracket V \rrbracket)}{!y\{x.\,y \leftrightarrow x\} \vdash y : !!(\llbracket U \rrbracket \otimes \llbracket V \rrbracket^\perp), x : ?(\llbracket U \rrbracket^\perp \parr \llbracket V \rrbracket)}$$

  Cut y of this with x' of P and we are done.

- If T is $U+V$, and both U and V are unlimited. First we apply lemma 9 on P and acquire $P_0 \vdash \mathcal{G}$ and $P_1 \vdash \Gamma, x : ?(\llbracket U \rrbracket^\perp \mathbin{\&} \llbracket V \rrbracket^\perp)$, and we have $\mathcal{D}_U$ defined as

$$\dfrac{\dfrac{\dfrac{y \leftrightarrow x \vdash y : \llbracket U \rrbracket, x : \llbracket U \rrbracket^\perp}{y[\textsc{l}].\,y \leftrightarrow x \vdash y : \llbracket U \rrbracket \oplus \llbracket V \rrbracket, x : \llbracket U \rrbracket^\perp}}{x'[\textsc{use}].\,xy[\textsc{l}].\,y \leftrightarrow x \vdash y : \llbracket U \rrbracket \oplus \llbracket V \rrbracket, x' : ?\llbracket U \rrbracket^\perp}}{\overline{?}x[x'].!y\{x'.\,x'[\textsc{use}].\,xy[\textsc{l}].\,y \leftrightarrow x\} \vdash y : !(\llbracket U \rrbracket \oplus \llbracket V \rrbracket), x : \llbracket U \rrbracket^\perp}$$

  and similarly for $\mathcal{D}_V$. Finally we have

$$\dfrac{\mathcal{D}_U \vdash y : !(\llbracket U \rrbracket \oplus \llbracket V \rrbracket), x : \llbracket U \rrbracket^\perp \qquad \mathcal{D}_V \vdash y : !(\llbracket U \rrbracket \oplus \llbracket V \rrbracket), x : \llbracket V \rrbracket^\perp}{x.\mathsf{case}\{\textsc{l}{:}\mathcal{D}_U, \textsc{r}{:}\mathcal{D}_V\} \vdash y : !(\llbracket U \rrbracket \oplus \llbracket V \rrbracket), x : \llbracket U \rrbracket^\perp \mathbin{\&} \llbracket V \rrbracket^\perp}$$

  Cut y of this with x' of P_1 then combine with P_0 and we are done.

- If T is $U \otimes V$, and both U and V are unlimited, we have

$$\dfrac{\dfrac{\dfrac{v \leftrightarrow v' \vdash v' : \llbracket V \rrbracket, v : \llbracket V \rrbracket^\perp \qquad u \leftrightarrow u' \vdash u' : \llbracket U \rrbracket, u : \llbracket U \rrbracket^\perp}{v'[u'].\,(v \leftrightarrow v' \mid u \leftrightarrow u') \vdash v' : \llbracket U \rrbracket \otimes \llbracket V \rrbracket, u : \llbracket U \rrbracket^\perp, v : \llbracket V \rrbracket^\perp}}{\begin{array}{c} u''[\textsc{use}].\,uv''[\textsc{use}].\,vv'[u'].\,(v \leftrightarrow v' \mid u \leftrightarrow u') \\ \vdash v' : \llbracket U \rrbracket \otimes \llbracket V \rrbracket, u'' : ?\llbracket U \rrbracket^\perp, v'' : ?\llbracket V \rrbracket^\perp \end{array}}}{\begin{array}{c} v(u).\,\overline{?}u[u''].\overline{?}v[v''].!v'\{u, v.\,u''[\textsc{use}].\,uv''[\textsc{use}].\,vv'[u'].\,(v \leftrightarrow v' \mid u \leftrightarrow u')\} \\ \vdash v' : !(\llbracket U \rrbracket \otimes \llbracket V \rrbracket), v : \llbracket U \rrbracket^\perp \parr \llbracket V \rrbracket^\perp \end{array}}$$

Cut v' of this with x of P, rename v to x and we are done.

□

2.D More Examples

2.D.1 Compare-And-Set

We now recover the example of CAS server/client in CSGV. We define the server-client protocol to be $T_S := !\mathbb{B}.!\mathbb{B}.?\mathbb{B}.\mathsf{end}_!$. The choice of $\mathsf{end}_?$ vs. $\mathsf{end}_!$ is purely driven by well-typedness.

$$\begin{aligned}
C_0 &:= \mathsf{let}\ x_d = \mathsf{send}\ \mathsf{ff}\ x_0\ \mathsf{in}\\
&\quad\ \mathsf{let}\ x_r = \mathsf{send}\ \mathsf{tt}\ x_d\ \mathsf{in}\\
&\quad\ \mathsf{let}\ (r, x') = \mathsf{recv}\ x_r\ \mathsf{in}\\
&\quad\ \mathsf{let}\ \_ = \mathsf{terminate}\ (\mathsf{send}\ r\ r0)\ \mathsf{in}\\
&\quad\ x'\\
C_1 &:= \ldots\\
\mathsf{clients} &:= \mathsf{let}\ x = \mathsf{fork}_x\ x_0.\, C_0\ \mathsf{in}\\
&\quad\ \mathsf{let}\ x = \mathsf{fork}_x\ x_1.\, C_1\ \mathsf{in}\\
&\quad\ \mathsf{eof}_x
\end{aligned}$$

$$\begin{aligned}
L &:= \mathsf{ff}\\
M &:= \mathsf{let}\ (exp, y') = \mathsf{recv}\ y\ \mathsf{in}\\
&\quad\ \mathsf{let}\ (des, y'') = \mathsf{recv}\ y'\ \mathsf{in}\\
&\quad\ \mathsf{if}\ exp = w\ \mathsf{then}\\
&\quad\ \mathsf{let}\ \_ = \mathsf{terminate}\ (\mathsf{send}\ \mathsf{tt}\ y'')\ \mathsf{in}\ \ des\\
&\quad\ \mathsf{else}\\
&\quad\ \mathsf{let}\ \_ = \mathsf{terminate}\ (\mathsf{send}\ \mathsf{ff}\ y'')\ \mathsf{in}\ \ w\\
N &:= z\\
\mathsf{server} &:= \mathsf{serve}\ y\{L, w.\, M, z.\, N\}
\end{aligned}$$

typed as:

$$\begin{aligned}
x_0 : T_S, r_0 : !\mathbb{B}.\mathsf{end}_? &\vdash C_0 : \mathsf{end}_! \\
x_1 : T_S, r_1 : !\mathbb{B}.\mathsf{end}_? &\vdash C_1 : \mathsf{end}_! \\
x : ¿T_S, r_0 : !\mathbb{B}.\mathsf{end}_?, r_1 : !\mathbb{B}.\mathsf{end}_? &\vdash \mathsf{clients} : \mathsf{end}_!
\end{aligned}$$

$$\begin{aligned}
&\vdash L : \mathbb{B} \\
w : \mathbb{B}, y : ?\mathbb{B}.?\mathbb{B}.?\mathbb{B}.\mathsf{end}_? &\vdash M : \mathbb{B} \\
z : \mathbb{B} &\vdash N : \mathbb{B} \\
y : ¡\overline{T_S} &\vdash \mathsf{server} : \mathbb{B}
\end{aligned}$$

where $\overline{T_S} = ?\mathbb{B}.?\mathbb{B}.!\mathbb{B}.\mathsf{end}_?$. Finally we have

$$r_0 : !\mathbb{B}.\mathsf{end}_?, r_1 : !\mathbb{B}.\mathsf{end}_? \vdash \mathsf{connect}(x.\,\mathsf{clients}; y.\,\mathsf{server}) : \mathbb{B}$$

2.D.2 List Shuffling

We use the racing behaviour of clients to shuffle a list. We define server/client protocol to be $T_S := !A.\mathsf{end}_!$, meaning each client sends a value of A and ends the session. Each clients are defined the same way: they simply take the value A from the environment and send it over the channel. Clients are forked by folding the list. The server simply receives values from clients and reforms the list.

$$\begin{aligned}
C &:= \mathsf{send}\ v\ x' \\
\mathsf{clients} &:= \mathsf{let}\ y\ =\ \mathsf{fold}_{¿T_S}\ x\ \lambda x.\,\lambda v.\,(\mathsf{fork}_x\ x'.\,C)\ l\ \mathsf{in} \\
&\ \mathsf{eof}_y
\end{aligned}$$

$$\begin{aligned}
L &:= \mathsf{nil} \\
M &:= \mathsf{let}\ (v, y')\ =\ \mathsf{recv}\ y\ \mathsf{in} \\
&\ \mathsf{let}\ \_\ =\ \mathsf{terminate}\ y'\ \mathsf{in} \\
&\ \mathsf{cons}\ v\ z' \\
N &:= z \\
\mathsf{server} &:= \mathsf{serve}\ y\{L, z'.\,M, z.\,N\}
\end{aligned}$$

and we have the following typing

$$\begin{aligned}
v : A, x' : T_S &\vdash C : \mathsf{end}_! & z' : [A], y : \overline{T_S} &\vdash M : [A] \\
l : [A], x : ¿T_S &\vdash \mathsf{clients} : \mathsf{end}_! & z : [A] &\vdash N : [A] \\
&\vdash L : [A] & y : ¡\overline{T_S} &\vdash \mathsf{server} : [A]
\end{aligned}$$

and finally we define shuffling as

$$l : [A] \vdash \mathsf{connect}(x.\,\mathsf{clients}; y.\,\mathsf{server}) : [A]$$

2.D.3 Merge Sort

Using fork-join, we can define parallel merge sort. We first have to assume general recursion (and therefore not expressible in the vanilla CSGV), and two functions on lists of A. split splits a list of A into several (supposedly two) lists, and merge merges several sorted lists into one list. isend returns tt if the list is empty or singleton.

$$\vdash \mathsf{split} : [A] \to [[A]] \qquad \vdash \mathsf{merge} : [[A]] \to [A] \qquad \vdash \mathsf{isend} : [A] \to \mathbb{B}$$

And we define merge sort as follows. We first check if the list l is too short to sort; if not we split the list and sort each of the sub-lists. The sorted sub-lists are collected into l' which we will merge. Note that the racing behaviour of client/server means l' could be any ordering, which does not matter for merge sort. For scnerios where it does matter, each sub-result should be accompanied by its index to get re-ordered.

$$\begin{aligned}\mathsf{sort}\ l := {} & \mathsf{if}\ \mathsf{isend}\ l\ \mathsf{then}\ l \\ & \mathsf{else} \\ & \mathsf{let}\ l' = \mathsf{fork\text{-}join}(\mathsf{sort}, \mathsf{cons}, \mathsf{nil}, (\mathsf{split}\ l))\ \mathsf{in} \\ & \mathsf{merge}\ l'\end{aligned}$$

which gives us

$$\vdash \mathsf{sort} : [A] \to [A]$$

2.D.4 Map-Reduce

The purpose of the map-reduce model is to transform input of type $[A]$ into output of type $[D]$ using two functions (using the functorial formulation given by Hinrichsen et al. [2019]). Take the example of counting the frequency of each word in an article which contains several paragraphs $[A]$. The *map* function f counts the frequency C of each word B in a paragraph. The *reduce* function g takes a word B and its frequency $[C]$ in all paragraphs and simply returns the word with the sum frequency $D := B \otimes C$. In the end we hope to get $[D]$.

$$f : A \to [B \otimes C] \qquad\qquad g : B \otimes [C] \to D$$

We first define parallelized $\mathsf{flatMap}$ with fork-join:

$$\vdash \mathsf{flatMap}_{A,B} := \lambda f.\, \lambda l.\, \mathsf{fork\text{-}join}(f, \mathsf{nil}, \mathsf{concat}, l) : (A \to [B]) \to [A] \to [B]$$

where concat is the standard function that concatenate two lists. Based on this we define $\mathsf{map\text{-}reduce}$:

$$\begin{aligned} f, g \vdash & \mathsf{map\text{-}reduce} := (\mathsf{flatMap}_{B\otimes[C],D}\ g) \circ \mathsf{group}_{B,C} \circ (\mathsf{flatMap}_{A,B\otimes C}\ f) \\ & : [A] \to [D]\end{aligned}$$

where $\vdash \mathsf{group}_{B,C} : [B \otimes C] \to [B \otimes [C]]$ is the standard function that groups list of pairs by their keys.

There is a notable difference between our version of map-reduce and the version in Hinrichsen et al. [2019] (and other related literatures). Usually a fixed number of threads (that usually corresponds to the number of cpu cores/nodes) are spawned, who will then repeatedly retrieve tasks from and send result to the main thread. In our version, however, the number of threads is the number of tasks, and each thread will handle one task only. The former approach seems lower-level, allowing optimizing the number of threads according to the hardware reality. Our language is higher-level, and it is up to the implementation to coordinate threads with cores/nodes. Implementing it at a lower-level seems to be difficult because of the linearity constraints.

2.D.5 Interleaving clients

Another interleaving clients example (but simpler than the beauty contest example) is one where each client submits a boolean to the server, who calculates the XOR of all the submissions and sends the result back to all clients. The internal protocol of the server, as well as the server interface, will be $T_S := ?\mathbb{B}.!\mathbb{B}.\mathsf{end}_?$. We define

$$
\begin{aligned}
L :=& \mathsf{let}\ w' = \mathsf{send}\ \mathsf{ff}\ w\ \mathsf{in} && \text{(send initial value)}\\
& \mathsf{let}\ (\_, w'') = \mathsf{recv}\ w'\ \mathsf{in} && \text{(recv the final value)}\\
& w'' \\
M :=& \mathsf{let}\ (s, z'') = \mathsf{recv}\ z'\ \mathsf{in} && \text{(recv the last value)}\\
& \mathsf{let}\ (b, y') = \mathsf{recv}\ y\ \mathsf{in} && \text{(recv the boolean from client)}\\
& \mathsf{let}\ s' = \mathsf{xor}(s, b)\ \mathsf{in} && \text{(calculate the xor)}\\
& \mathsf{let}\ w' = \mathsf{send}\ s'\ w\ \mathsf{in} && \text{(send to next worker process)}\\
& \mathsf{let}\ (f, w'') = \mathsf{recv}\ w'\ \mathsf{in} && \text{(recv the final boolean)}\\
& \mathsf{let}\ \_ = \mathsf{terminate}\ (\mathsf{send}\ f\ z'')\ \mathsf{in} \\
& && \text{(send the final to previous worker process)}\\
& \mathsf{let}\ \_ = \mathsf{terminate}\ (\mathsf{send}\ f\ y')\ \mathsf{in} && \text{(send the final to client)}\\
& w'' \\
N :=& \mathsf{let}\ (f, z') = \mathsf{recv}\ z\ \mathsf{in} && \text{(recv the final value)}\\
& \mathsf{let}\ z'' = \mathsf{send}\ f\ z'\ \mathsf{in} && \text{(simply send it back)}\\
& \mathsf{terminate}\ z'' \\
\mathsf{server} :=& \mathsf{serve}\ y\{\mathsf{inv}_w(L), z'.\, \mathsf{inv}_w(M), z.\, N\}
\end{aligned}
$$

where $\vdash \mathsf{xor} : \mathbb{B} \otimes \mathbb{B} \to \mathbb{B}$ is the standard xor function on booleans.

$$\frac{w : \overline{T_S} \vdash L : \mathsf{end}_! \qquad w : \overline{T_S}, z' : T_S, y : T_S \vdash M : \mathsf{end}_! \qquad z : T_S \vdash N : \mathsf{Unit}}{\mathsf{xor}, y : ¡T_S \vdash \mathsf{server} : \mathsf{Unit}}$$

We omit defining the clients as they will be very similar to the ones in previous examples.

2.D.6 Symbol Generator

Another simple scenario is where server acts like a generator of unique symbols (essentially natural numbers $\mathbb{N}$) and clients race to acquire those symbols. The server protocol is $!\mathbb{N}.\mathsf{end}_?$, meaning the server simply sends an number to the client and ends the session; the server internal state is $\mathbb{N}$.

$$\begin{aligned}
L &:= \mathsf{zero} && \text{(starts with zero)}\\
M &:= \mathsf{let}\ \_ = \mathsf{terminate}\ (\mathsf{send}\ z'\ y)\ \mathsf{in} && \text{(send the counter to client)}\\
&\quad\ \mathsf{succ}\ z' && \text{(increase the counter)}\\
N &:= z && \text{(output the final counter)}\\
\mathsf{server} &:= \mathsf{serve}\ y\{L, z'.\, M, z.\, N\}
\end{aligned}$$

typed as

$$\frac{\vdash L : \mathbb{N} \qquad z' : \mathbb{N}, y : !N.\mathsf{end}_? \vdash M : \mathbb{N} \qquad z : \mathbb{N} \vdash N : \mathbb{N}}{y : ¡(!\mathbb{N}.\mathsf{end}_?) \vdash \mathsf{server} : N}$$

We omit defining the clients as they would be similar to previous ones; but we note that it is impossible for a process to act as multiple clients and aggregate two symbols. The reason is that informally speaking, in our system clients are not allowed to communicate with each other besides via the server as indicated by the functor. More concretely, supposed we are to define a process acting as multiple clients, it would be typed as $\Gamma, x_0 : ?N.\mathsf{end}_?, x_1 : ?N.\mathsf{end}_? \vdash K : T$; but there is no way in CSGV to combine x_0 and x_1.

Chapter 3

Concurrent Effects in Linear Logic

This chapter is based on Qian et al. [2022], but I removed part of section 3.1 that is general to classical processes, which has been incorporated into section 1.1. I also removed part of section 3.8 that is general to classical processes, which has been incorporated into section 1.3. I also reformatted the text to fit the new paper size.

Qian et al. [2022] requires a small but fundamental change on top of Montesi and Peressotti [2021]. It was omitted in Qian et al. [2022] due to oversight. In this chapter, we describe this change in section 3.2. Most meta-theoretic results still hold and are proved from scratch in section 3.6.

3.1 Introduction

Among concurrency features missing in Classical Process, the arguably most recognizable one is probably that of shared effects, which can be characterized as follows. Each process emits effects in sequence, in the sense that emissions of later effects depend on the results of earlier effects; moreover, multiple processes can emit effects in a sharing manner.

Many have tried to model shared effects, each with their shortcomings. Balzer and Pfenning [2017] formulates a type system stratified into linear and non-linear layers, which are bridged by locks: processes race to acquire locks guarding the linear layer where actual accesses are performed. Unfortunately, the system introduces deadlocks unless one adopts a more complex type system [Balzer et al., 2019] that is not logically justified. Rocha and Caires [2021] extends Wadler [2014] with cells that can be shared among racing processes. However, they can only store positive values, which is a severe constraint in the context of session-based concurrency: they cannot store sessions. Also, their typing rules are ad-hoc for positive cells instead of driven by logic.

Towards concurrent classical effects

This paper is the first attempt to model concurrent classical effects based on classical linear logic, in the hope of finding a theory that is general, simple, and logical. We now outline some of the key ideas of our approach.

First recall that $\otimes$ represents disjointness while $\parr$ represents connectedness. We start with some simple candidates of modelling (sequences of) effects. Let E be the type of an effect. First we consider $E \otimes E \otimes \cdots$, which cannot work because $\otimes$ means effect emissions of effects are disjoint from each other. Consequently, results of previous effects cannot be passed to the emissions of following effects, which breaks sequentiality. We then consider $E \parr E \parr \cdots$, which cannot work either, because the state on the dual side would have type $E^\perp \otimes E^\perp \otimes \cdots$, which means effect handlings are disjoint from each other. Consequently, handling of earlier effects cannot influence the handling of later effects, which means the handling is stateless.

With the two simple candidates ruled out, we move to more complex ones. Consider

$$A \otimes (B^\perp \parr (A \otimes (B^\perp \parr (\cdots$$

where an effect E is decomposed into two parts: sending request A and receiving response B (and thus negated). Only the response is connected to the rest of the emitting via $\parr$, which is sufficient for sequentiality. Dually, the handling would have type

$$A^\perp \parr (B \otimes (A^\perp \parr (B \otimes (\cdots$$

where the response is disjoint from the rest of the handling via $\otimes$. This is fine, because we do not expect either of them to influence the other. Note that the received request A can influence both the response B to be sent and the rest of the handling.

The above definition also admits sharing by the simple linear logic implication (subscripts only for identification):

$$\begin{aligned}&(A_0 \otimes B_0^\perp \parr \cdots \parr C) \otimes (A_1 \otimes B_1^\perp \parr \cdots \parr D)\\ \to &A_0 \otimes B_0^\perp \parr ((\cdots \parr C) \otimes (A_1 \otimes B_1^\perp \parr \cdots \parr D))\end{aligned}$$

where the lhs (left-hand side) has two disjoint emission sequences each ready to emit an effect, while the right-hand side (rhs) has an effect already propagated outside of the first sequence. Intuitively, the first sequence won the race. There is a symmetric version where the second sequence wins. Note that the implication is one-directional—the outcome of the race cannot be reversed.

The above informal definition leads to a formal one based on least and greatest fixed point in linear logic [Baelde, 2012, Lindley and Morris, 2016], from which typing rules and a transition semantics can be derived. We call the resulting system CELL. Good properties such as deadlock freedom and session fidelity follow. In particular, the handling of effects requires an internal state type S and a process of type (subscripts are only for identification):

$$(A \otimes S_{\mathsf{old}}) \multimap (B \otimes S_{\mathsf{new}}) \tag{3.1}$$

We can also let $T_{\mathsf{new}} = S_{\mathsf{old}}^{\perp}$ and $T_{\mathsf{old}} = S_{\mathsf{new}}^{\perp}$ and rewrite the above as

$$A \multimap (B \multimap T_{\mathsf{old}}) \multimap T_{\mathsf{new}} \tag{3.2}$$

The formulas remind one of algebraic effects [Pretnar, 2015], where section 3.1 reminds one of runners [Uustalu, 2015] and section 3.1 of handlers [Pretnar and Plotkin, 2013]. The two are well-known to be dual [Plotkin and Power, 2008, Power and Shkaravska, 2004], and thus not suprising [Hasegawa, 2002, §8] to collide in linear logic. We can utilize this fact and express both in our system.

Contributions

- We extend πLL[Montesi and Peressotti, 2021] with effects and obtain CELL: a process calculus with linear logic types and effects. Processes can emit effects in sequence and multiple processes can race for effects. Effects are handled by another (dual) process. In addition, we introduce a primitive for inter-process synchronization via effects at the cost of livelocks.

- We give a labeled transition system semantics for effects and prove meta-theoretic results such as erasure, session fidelity, deadlock-freedom, and the partial diamond property.

- We introduce CEGV a functional language with session-typed communication primitives and effects, based on GV [Wadler, 2014]. Both handlers and runners are available in the system, and the duality is made explicit. Thin translations to CELL are given. We give examples such as effect translation and escaping instances. Racing is also lifted to CEGV, which is used to express concurrency examples such as worker-pool servers and dining philosophers.

- Based on our LTS, we prove several desired bisimilarity results. For example, that handling of different effects does not interfere; that the monadic interface for effectful computation is indeed monadic; and that spawned processes race with their continuation.

3.2 Base system

In this section we briefly recall the basics of πLL Montesi and Peressotti [2021]. For conciseness, we omit from this sections features related to replicable processes/exponentials and polymorphism/logical quantifiers since these are orthogonal to the developments of this work and can be readily incorporated.

Processes Programs in πLL are processes (P,Q,R). Processes communicate over binary sessions using names (x,y,z) representing session endpoints. The process terms of πLL are given by the following grammar.

$P, Q, R := x[y].\,P$	*output y on x and continue as P*
$\mid x(y).\,P$	*input y on x and continue as P*
$\mid x[].\,P$	*output (empty message) on x and continue as P*
$\mid x().\,P$	*input (empty message) on x and continue as P*
$\mid x[\mathrm{L}].\,P$	*select left (output label* L*) on x and continue as P*
$\mid x[\mathrm{R}].\,P$	*select right (output label* R*) on x and continue as P*
$\mid y.\mathsf{case}\{\mathrm{L}{:}P, \mathrm{R}{:}Q\}$	*offer on x a choice to continue as P and Q*
$\mid \mathbf{0}$	*terminated process*
$\mid P \mid Q$	*parallel composition of P and Q*
$\mid \nu xy\, P$	*session with endpoints x and y in P*
$\mid x \leftrightarrow y$	*forwarding of x and y*

Term $x[y].\,P$ allocates a new endpoint with fresh name y (bound in the continuation P), outputs a connection request for y over x, and then proceeds as P. Dually, $x(y).\,P$ allocates a new endpoint y (bound in P) and awaits for a connection request on x before continuing as P. The result of synchronising these actions is the creation of a new session. Terms $x[].\,P$ and $x().\,P$ respectively output and input messages with no content—essentially a handshake. Terms $x[\mathrm{L}].\,P$, $x[\mathrm{R}].\,P$, and $x.\mathsf{case}\{\mathrm{L}{:}P, \mathrm{R}{:}Q\}$ represent the selection and offering of a binary choice: $x[\mathrm{L}].\,P$ and $x[\mathrm{R}].\,P$ output over x the labels L and R before continuing as P; dually, $x.\mathsf{case}\{\mathrm{L}{:}P, \mathrm{R}{:}Q\}$ continues as P when the it receives L over x and as Q when it receives R.

In the reminder, we use π to range over term prefixes $x(y)$, $x[y]$, $x()$, $x[]$, $x(\mathrm{L})$, $x(\mathrm{R})$. We write $\mathrm{FN}(P)$, $\mathrm{BN}(P)$, and $\mathrm{N}(P)$ for the set of free, bound, and all endpoint names in P, respectively, and likewise for prefixes. We write $P =_\alpha Q$ if P and Q are α-equivalent.

Types and environments Types in πLL (A, B, ...) are propositions in classical linear logic and are interpreted as protocols for single endpoints.

$A, B := A \otimes B$	*send A, continue as B*	$\mid A \parr B$	*receive A, continue as B*
$\mid \mathbf{1}$	*send close, unit for $\otimes$*	$\mid \bot$	*receive close, unit for $\parr$*
$\mid A \oplus B$	*select A or B*	$\mid A \mathbin{\&} B$	*offer A or B*

$$\textsc{HMix0}\ \frac{}{\mathbf{0} \vdash \emptyset} \qquad \textsc{HMix}\ \frac{P \vdash \mathcal{G} \qquad Q \vdash \mathcal{H}}{P \mid Q \vdash \mathcal{G} \mid \mathcal{H}} \qquad \textsc{Cut}\ \frac{P \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : A^{\perp}}{\nu xy\, P \vdash \mathcal{G} \mid \Gamma, \Delta}$$

$$\textsc{Ax}\ \frac{}{x \leftrightarrow y \vdash x : A^{\perp}, y : A} \qquad \textsc{Tensor}\ \frac{P \vdash \Gamma, y : A \mid \Delta, x : B}{x[y].\, P \vdash \Gamma, \Delta, x : A \otimes B} \qquad \textsc{One}\ \frac{P \vdash \emptyset}{x[].\, P \vdash x : \mathbf{1}}$$

$$\textsc{Par}\ \frac{P \vdash \Gamma, y : A, x : B}{x(y).\, P \vdash \Gamma, x : A \parr B} \qquad \textsc{Bot}\ \frac{P \vdash \Gamma}{x().\, P \vdash \Gamma, x : \bot} \qquad \textsc{PlusL}\ \frac{P \vdash \Gamma, x : A}{x[\textsc{l}].\, P \vdash \Gamma, x : A \oplus B}$$

$$\textsc{PlusR}\ \frac{P \vdash \Gamma, x : B}{x[\textsc{r}].\, P \vdash \Gamma, x : A \oplus B} \qquad \textsc{With}\ \frac{P \vdash \Gamma, x : A \qquad Q \vdash \Gamma, x : B}{x.\mathsf{case}\{\textsc{l}{:}P, \textsc{r}{:}Q\} \vdash \Gamma, x : A \,\&\, B}$$

Figure 1: πLL, typing rules (multiplicative and additive fragment) Montesi and Peressotti [2021].

Types on the left-hand column are for output actions and types on the right-hand for inputs. Under this interpretation, the notion of duality of linear logic relates the matching input/output actions of two endpoints interacting over a shared session (connectives on the same row are respective duals, e.g.,$\otimes$ and $\parr$). We write $A^{\perp}$ for the dual of A.

Typing environments (Γ,Δ,...) associate endpoint names to types and hyperenvironments ($\mathcal{G}$, $\mathcal{H}$, ...) are unordered collections of environments that do not share endpoint names:

$$\Gamma, \Delta := x_1 : A_1, \ldots, x_n : A_n \qquad \mathcal{G}, \mathcal{H} := \Gamma_1 \mid \cdots \mid \Gamma_n.$$

Intuitively, endpoints within the same environment can have sequential or concurrent implementations whereas endpoints from different environments are guaranteed to have parallel, independent implementations. We write $\cdot$ and $\mathbf{1}$ for the empty environment and the empty hyperenvironments, respectively; $\mathrm{N}(\Gamma)$ and $\mathrm{N}(\mathcal{G})$ for the set of endpoint names in Γ and $\mathcal{G}$, respectively.

Typing Typing judgements have form $P \vdash \mathcal{G}$ and indicate that the process uses its endpoints as specified by the hyperenvironment. The rules for deriving these judgements are reported in fig. 1. These rules associate types to endpoint names by looking at how they are used in process terms. We refer the interested reader to Montesi and Peressotti [2021] for more details on these rules.

We range over typing derivations with letters $\mathcal{D}$, $\mathcal{E}$. We write $\mathit{proc}(\mathcal{D})$ and $\mathit{env}\mathcal{D}$ for the process and typing environment in the conclusion of $\mathcal{D}$, respectively.

Like the internal π-calculus, πLL recovers the π-calculus primitive for outputting free names as syntactic sugar $x\langle y\rangle.\, P := x[z].\,(y \leftrightarrow z \mid P)$. In the remainder we use also abbreviate $\mathbf{0}_x := x[].\,\mathbf{0}$ and $P{}^{x}_{y}\!\otimes^{w}_{v}Q := w(x).\, v[y].\,(P \mid Q)$. This syntactic sugar induces the (derivable) typing rules.

$$\frac{P \vdash \Gamma, x : B}{x\langle y\rangle.\, P \vdash \Gamma, x : A^{\perp} \otimes B, y : A} \qquad \frac{}{\mathbf{0}_x \vdash x : \mathbf{1}}$$

$$\frac{P \vdash x : A, y : B \qquad Q \vdash w : C, v : D}{P{}^{x}_{y}\!\otimes^{w}_{v}Q \vdash w : A \parr C, v : B \otimes D}$$

Operational Semantics The dynamics of πLLprocesses is specified as a labelled transition system (lts) for typing derivations following the application of the Structural Operational Semantics (SOS) style to typing derivations as originally proposed by Montesi and Peressotti [2018]. Under this approach, we view:

1. typing rules as operations of a (sorted) signature;
2. typing derivations as terms generated by this signature;
3. transformations of derivations as transitions;
4. and a specifications of rules for deriving these transformations as an SOS specification.

As an example consider a derivation terminating with an application of Bot, like the one displayed below on the left of the transition. Under this approach, rule Bot is regarded as a unary operator applied to the derivation $\mathcal{D}$. This operator corresponds to $x().\,(-)$ in the syntax of processes which, in this example, takes P as its continuation (the proof term of $\mathcal{D}$). In the π-calculus, the semantics of terms of this form is given by transitions where the target (or derivative) is the operation argument (the continuation) and the transition label is the prefix representing the action. This translates to the following transition rule for Bot.

$$\frac{\begin{array}{c}\mathcal{D}\\ P \vdash \Gamma\end{array}}{x().\, P \vdash \Gamma, x : \perp}\ \textsc{Bot} \quad \xrightarrow{x()} \quad \begin{array}{c}\mathcal{D}\\ P \vdash \Gamma\end{array}$$

In the sequel, we will omit names of derivations in the presentation of transition rules to save space.

Following the same methodology, Montesi and Peressotti [2021] define an SOS specification for πLL–which we include in section 3.A. From this SOS specification and lts of derivations, they systematically derive two additional

SOS specifications and transition systems: one for process terms and one for type environments. The first is obtained by erasing all information about types and the second by erasing process terms. Applying this procedure is applied to the transition rule for Bot yields the following rules for processes and typing environments.

$$x().\,P \xrightarrow{x()} P \qquad\qquad \Gamma, x : \bot \xrightarrow{x()} \Gamma$$

We report the SOS specification for the lts of processes and environments in fig. 2, fig. 3—we denote that sets S and S' are disjoint by writing $S \# S'$.

We make one small but fundamental change over Montesi and Peressotti [2021]. In the original work, labels could be τ, an action, or two actions in parallel. In our system, labels are generalized to multisets of parallel actions, where τ stands for the empty set. This allows three or more parallel actions in a label, and enables the rule BoCW in section 3.4. The change preserves most properties including erasure and session fidelity, which are proved from scratch again in section 3.C. Some properties such as serialization might still hold after reformulation, which we leave to future works.

Coherence between these systems is captured by properties of *erasure* and *session fidelity*. Erasure states that the semantics of processes does not rely on runtime information about their types.

Theorem 17 (Erasure)**.** *For any derivation $\mathcal{D}$ and label l:*

- *if $\mathcal{D} \xrightarrow{l} \mathcal{D}'$, then $proc(\mathcal{D}) \xrightarrow{l} proc(\mathcal{D}')$;*
- *if $proc(\mathcal{D}) \xrightarrow{l} P'$, then $\mathcal{D} \xrightarrow{l} \mathcal{D}'$ for some $proc(\mathcal{D}') = P'$.*

Session fidelity states that processes perform only actions allowed by their typing environment.

Theorem 18 (Session Fidelity)**.** *If $P \vdash \mathcal{G}$ and $P \xrightarrow{l} P'$, then $P' \vdash \mathcal{G}'$ and $\mathcal{G} \xrightarrow{l} \mathcal{G}'$ for some $\mathcal{G}'$.*

πLL supports the expected notion of bisimulation.

Definition 4. A relation $\mathcal{R} \subseteq S \times T$ is a *strong bisimulation* for two lts $(S, L, \rightarrow)$ and $(T, L, \rightarrow)$ when $s \,\mathcal{R}\, t$ implies that:

- if $s \xrightarrow{l} s'$ then $t \xrightarrow{l} t'$ for some t' such that $s' \,\mathcal{R}\, t'$;
- if $t \xrightarrow{l} t'$ then $s \xrightarrow{l} s'$ for some s' such that $s' \,\mathcal{R}\, t'$.

Strong bisimilarity is the largest relation $\sim \subseteq S \times T$ that is a strong bisimulation. The *saturation* of an lts $(S, L, \rightarrow)$ is the $(S, L, \Rightarrow)$ where $\Rightarrow$ is the smallest relation such that: $s \overset{\tau}{\Rightarrow} s$ for all $s \in S$ and if $s_1 \overset{\tau}{\Rightarrow} s_2$, $s_2 \xrightarrow{l} s_3$,

$$\pi.P \xrightarrow{\pi} P \qquad x.\mathsf{case}\{\textsc{l}{:}P, \textsc{r}{:}Q\} \xrightarrow{x(\textsc{l})} P \qquad x.\mathsf{case}\{\textsc{l}{:}P, \textsc{r}{:}Q\} \xrightarrow{x(\textsc{r})} Q$$

$$x \leftrightarrow y \xrightarrow{x\leftrightarrow y} \mathbf{0} \qquad x \leftrightarrow y \xrightarrow{y\leftrightarrow x} \mathbf{0}$$

$$\begin{array}{l}\textsc{Par0}\\ \dfrac{P \xrightarrow{l} P' \qquad \textsc{Bn}(l) \# \textsc{Fn}(Q)}{P \mid Q \xrightarrow{l} P' \mid Q}\end{array} \qquad \begin{array}{l}\textsc{Par1}\\ \dfrac{Q \xrightarrow{l} Q' \qquad \textsc{Bn}(l) \# \textsc{Fn}(Q)}{P \mid Q \xrightarrow{l} P \mid Q'}\end{array}$$

$$\begin{array}{l}\textsc{Syn}\\ \dfrac{P \xrightarrow{l} P' \qquad Q \xrightarrow{l'} Q' \qquad \textsc{Bn}(l \mid l') \# \textsc{Fn}(P \mid Q)}{P \mid Q \xrightarrow{l|l'} P' \mid Q'}\end{array}$$

$$\begin{array}{l}\text{AEQ}\\ \dfrac{P =_\alpha Q \qquad Q \xrightarrow{l} Q'}{P \xrightarrow{l} Q'}\end{array} \qquad \begin{array}{l}\textsc{Res}\\ \dfrac{P \xrightarrow{l} P' \qquad x, y \notin \mathrm{N}(l)}{\nu xy\, P \xrightarrow{l} \nu xy\, P}\end{array} \qquad \begin{array}{l}\textsc{Link}\\ \dfrac{P \xrightarrow{y\leftrightarrow z} P'}{\nu xy\, P \xrightarrow{\tau} P'\{x/z\}}\end{array}$$

$$\begin{array}{l}\textsc{Tensor-Par}\\ \dfrac{P \xrightarrow{x[x']|y(y')} P'}{\nu xy\, P \xrightarrow{\tau} \nu xy\, \nu x'y'\, P'}\end{array} \qquad \begin{array}{l}\textsc{One-Bot}\\ \dfrac{P \xrightarrow{x[]|y()} P'}{\nu xy\, P \xrightarrow{\tau} P'}\end{array} \qquad \begin{array}{l}\textsc{Com}\\ \dfrac{P \xrightarrow{x[\mu]|y(\mu)} P' \qquad \mu \in \{\textsc{l}, \textsc{r}\}}{\nu xy\, P \xrightarrow{\tau} \nu xy\, P'}\end{array}$$

Figure 2: πLL, transition rules for processes Montesi and Peressotti [2021].

$$x : \mathbf{1} \xrightarrow{x[]} \emptyset \qquad \Gamma, x : \bot \xrightarrow{x()} \Gamma \qquad \Gamma, \Delta, x : A \otimes B \xrightarrow{x[x']} \Gamma, x : B \mid \Delta, x' : A$$

$$\Gamma, x : A \parr B \xrightarrow{x(x')} \Gamma, x : B, x' : A$$

$$\Gamma, x : A \oplus B \xrightarrow{x[\text{L}]} \Gamma, x : A \qquad \Gamma, x : A \,\&\, B \xrightarrow{x(\text{L})} \Gamma, x : A$$

$$\Gamma, x : A \oplus B \xrightarrow{x[\text{R}]} \Gamma, x : A \qquad \Gamma, x : A \,\&\, B \xrightarrow{x(\text{R})} \Gamma, x : A$$

$$x : A^{\bot}, y : A \xrightarrow{x \leftrightarrow y} \emptyset$$

$$\mathcal{G} \mid \Gamma \xrightarrow{\tau} \mathcal{G} \mid \Gamma \qquad \frac{\mathcal{G} \xrightarrow{l} \mathcal{G}'}{\mathcal{G} \mid \mathcal{H} \xrightarrow{l} \mathcal{G}' \mid \mathcal{H}}\ \textsc{Par0} \qquad \frac{\mathcal{H} \xrightarrow{l} \mathcal{H}'}{\mathcal{G} \mid \mathcal{H} \xrightarrow{l} \mathcal{G} \mid \mathcal{H}'}\ \textsc{Par1}$$

$$\frac{\mathcal{G} \xrightarrow{l} \mathcal{G}' \qquad \mathcal{H} \xrightarrow{l'} \mathcal{H}'}{\mathcal{G} \mid \mathcal{H} \xrightarrow{l \mid l'} \mathcal{G}' \mid \mathcal{H}'}\ \textsc{Syn}$$

Figure 3: πLL, transition rules for typing environments Montesi and Peressotti [2021].

and $s_3 \overset{\tau}{\Rightarrow} s_4$, then $s_1 \overset{l}{\Rightarrow} s_4$. A relation $\mathcal{R} \subseteq S \times T$ is a *bisimulation* for $(S, L, \to)$ and $(T, L, \to)$ when $\mathcal{R}$ is a strong bisimulation for their saturations. *Bisimilarity* is the largest relation $\approx \subseteq S \times T$ that is a bisimulation.

We write $P \xrightarrow{l}\approx Q$ if $P \xrightarrow{l} P'$ and $P' \approx Q$ for some P'.

Interestingly, erasure establishes a strong bisimulation between derivations and processes, and session fidelity establishes a strong simulation from processes to environments.

3.3 Classical Effects

This section extends the base system with effects. Since we work in classical linear logic, effects have to conform to duality. Thus we introduce two dual connectives $\boxbox$ and $\Diamonddot$; the former represents an emitter that emits effects, while the latter represents a coemitter that handles effects.

Recall that in the Introduction, the emitter was informally parameterized by a pair: a *request type* A and *response type* B. It is natural to generalize the emitter to support multiple kinds of effects. To this end, we introduce the notion of an effect environment, which is a mapping from *effect names* (such as i) to a pair of request and response types.

Effect environments are generated by the following grammar; $\bullet$ is empty effect environment.

$$\Theta ::= \bullet \mid \mathsf{i} : (A : B), \Theta$$

The dual $\Theta^\perp$ of an effect environment Θ is defined by dualising each binding $\mathsf{i} : (A : B)$ to $\mathsf{i} : (A^\perp : B^\perp)$.

We now set out to capture formally the (co)inductive nature of the emitter and the coemitter as informally exhibited in Introduction, by defining some functors of which we will take the least and greatest fixed points [Baelde, 2012]. For an effect environment Θ and some type C, we define F_C^Θ and G_C^Θ by induction on Θ:

$$\begin{aligned} F_C^\bullet(X) &:= C & \qquad F_C^{\mathsf{i}:(A:B),\Theta}(X) &:= F_C^\Theta(X) \oplus (A \otimes B \parr X) \\ G_C^\bullet(X) &:= C & \qquad G_C^{\mathsf{i}:(A:B),\Theta}(X) &:= G_C^\Theta(X) \mathbin{\&} (A \parr B \otimes X) \end{aligned}$$

Note that F_C^Θ is dual to $G_{C^\perp}^{\Theta^\perp}$, in the sense that $(F_C^\Theta(X))^\perp = G_{C^\perp}^{\Theta^\perp}(X^\perp)$. We can now define

$$\boxbox^\Theta C := \mu F_C^\Theta \qquad \text{(emitter)} \qquad\qquad \Diamonddot^\Theta C := \nu G_C^\Theta \qquad \text{(coemitter)}$$

The intuition is that $\boxbox^\Theta C$ is allowed to emit effects in Θ and returns C, while $\Diamonddot^\Theta C$ is capable of handling effects in Θ and returns C. We further specify the duality

$$(\boxbox^\Theta C)^\perp = \Diamonddot^{\Theta^\perp} C^\perp \qquad\qquad (\Diamonddot^\Theta C)^\perp = \boxbox^{\Theta^\perp} C^\perp$$

BoW
$$\frac{P \vdash \Gamma, x : C}{\boxdot x[].\, P \vdash \Gamma, x : \boxdot^{\Theta} C}$$

BoA
$$\frac{P \vdash \Delta, a : A \mid \Gamma, x : B \parr \boxdot^{\Theta} C \qquad \mathsf{i} : (A : B) \in \Theta}{\boxdot x^{\mathsf{i}}[a].\, P \vdash \Delta, \Gamma, x : \boxdot^{\Theta} C}$$

Di
$$\frac{\begin{array}{c} P \vdash \Gamma, i : S \\ \forall \mathsf{i} : (A : B) \in \Theta.\ Q^{\mathsf{i}} \vdash z : S^{\perp}, a' : A, z' : B \otimes S \qquad R \vdash f : S^{\perp}, y : C \end{array}}{\Diamonddot y\{i.\, P, za'z'.\, Q, f.\, R\} \vdash \Gamma, y : \Diamonddot^{\Theta} C}$$

Figure 4: Effects, typing rules

The duality together with Cut means that an emitter and a coemitter can interact if the former is allowed to emit only the effects that the latter is capable of handling, a.k.a. effect safety. Note that they have dual views of the effect environment, since whatever is 'sent' from the emitter will be 'received' by the coemitter, and vice versa. Similarly, their return types must be dual as well.

The typing rules of effects are given in fig. 4, all of which directly derived from the fixed point rules of Baelde [2012], Lindley and Morris [2016]. We proceed to comment on each of the rules. Weakening BoW gives an emitter without any effect. Absorption BoA gives an emitter with a i-effect of request A and response B, and the continuing emitter $\boxdot^{\Theta} C$. Rule Di constructs a coemitter: given the *internal state type* S, process P provides the initial internal state at i; for each effect $\mathsf{i} : (A : B) \in \Theta$ we have Q^{i} providing a response and new internal state at z' given old one at z and request at a'; finally, R takes the final internal state at f and terminates the handler by returning at y. Note the distinction between the session name i (in math italics) and the effect name i (in sans serif) Also note that Q^{i} corresponds to section 3.1 and section 3.1 in the introduction.

The LTS of effect is given in fig. 5. For clarify we also list the LTS projected to processes in fig. 6 and environments in fig. 3.B.1 in section 3.B. We will comment on fig. 5. An emitter has internal choice as it knows whether to end or to emit, depending on whether it is constructed by BoW or BoA. In BoWW, the emitter signals the label $\boxdot x[]$, indicating it will no longer emit effects. In BoAA, the emitter signals the label $\boxdot x^{\mathsf{i}}[a]$, indicating an emission of i-effect and request at a. In both cases, it transitions into the continuation P.

On the other hand, a coemitter relies on external choice, depending on which it can either end or handle. In DiW, the coemitter signals the label $\Diamonddot y()$, indicating that it will no longer take effects, and transitions into the return value C by connecting initialization and finalization. In DiA, the coemitter signals the label $\Diamonddot y^{\mathsf{i}}(a')$, indicating handling of a i-effect and

$$\boxdot x[].\,P \vdash \Gamma, x : \boxdot^{\Theta} C \xrightarrow[\textsc{BoWW}]{\boxdot x[]} P \vdash \Gamma, x : C$$

$$\boxdot x^{\mathsf{i}}[a].\,P \vdash \Delta, \Gamma, x : \boxdot^{\Theta, \mathsf{i}:(A:B)} C \xrightarrow[\textsc{BoAA}]{\boxdot x^{\mathsf{i}}[a]} P \vdash \Delta, a : A \mid \Gamma, x : B \parr \boxdot^{\Theta, \mathsf{i}:(A:B)} C$$

$$\diamonddot y\{i.\,P, za'z'.\,Q, f.\,R\} \vdash \Gamma, y : \diamonddot^{\Theta} C \xrightarrow[\textsc{DiW}]{\diamonddot y()} \nu if\,(P \mid R) \vdash \Gamma, y : C$$

$$\frac{\begin{array}{l} P \vdash \Gamma, i : S \\ \forall \mathsf{i} : (A : B) \in \Theta.\, Q^{\mathsf{i}} \vdash z : S^{\perp}, a' : A, z' : B \otimes S \\ R \vdash f : S^{\perp}, y : C \end{array}}{\diamonddot y\{i.\,P, za'z'.\,Q, f.\,R\} \vdash \Gamma, y : \diamonddot^{\Theta} C} \xrightarrow[\textsc{DiA}]{\diamonddot y^{\mathsf{i}}(a)}$$

$$\frac{\begin{array}{l} P \vdash \Gamma, i : S \\ Q^{\mathsf{i}} \vdash z : S^{\perp}, a' : A, z' : B \otimes S \\ (b \leftrightarrow b')\,{}^{b'}_{b}\!\otimes^{i'}_{y} T \vdash i' : B^{\perp} \parr S^{\perp}, y : B \otimes \diamonddot^{\Theta} C \end{array}}{\nu i'z'\,\nu iz\,(Q^{\mathsf{i}} \mid P \mid (b \leftrightarrow b')\,{}^{b'}_{b}\!\otimes^{i'}_{y} T) \vdash \Gamma, a' : A, y : B \otimes \diamonddot^{\Theta} C}\ \textsc{Cut}$$

where $T := \diamonddot y\{i.\,i' \leftrightarrow i, za'z'.\,Q, f.\,R\} \vdash i' : S^{\perp}, y : \diamonddot^{\Theta} C$

BoW-DiW

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x : \boxdot^{\Theta} C \mid \Delta, y : \diamonddot^{\Theta^{\perp}} C^{\perp} \xrightarrow{\boxdot x[] \mid \diamonddot y()} P' \vdash \mathcal{G} \mid \Gamma, x : C \mid \Delta, y : C^{\perp}}{\nu xy\, P \vdash \mathcal{G} \mid \Gamma, \Delta \xrightarrow{\tau} \nu xy\, P' \vdash \mathcal{G} \mid \Gamma, \Delta}$$

BoA-DiA

$$\frac{\begin{array}{c} P \vdash \mathcal{G} \mid \Gamma_0, \Gamma_1, x : \boxdot^{\Theta} C \mid \Delta, y : \diamonddot^{\Theta^{\perp}} C^{\perp} \xrightarrow{\boxdot x^{\mathsf{i}}[a] \mid \diamonddot y^{\mathsf{i}}(a')} \\ P' \vdash \mathcal{G} \mid \Gamma_0, a : A \mid \Gamma_1, x : B \parr \boxdot^{\Theta} C \mid \Delta, a' : A^{\perp}, y : B^{\perp} \otimes \diamonddot^{\Theta^{\perp}} C^{\perp} \\ \mathsf{i} : (A : B) \in \Theta \end{array}}{\nu xy\, P \vdash \mathcal{G} \mid \Gamma_0, \Gamma_1, \Delta \xrightarrow{\tau} \nu aa'\,\nu xy\, P' \vdash \mathcal{G} \mid \Gamma_0, \Gamma_1, \Delta}$$

Figure 5: Effects, transition rules for derivations

$$\boxdot x[].\, P \xrightarrow[\textsc{BoWW}]{\boxdot x[]} P \qquad \boxdot x^{\mathsf{i}}[a].\, P \xrightarrow[\textsc{BoAA}]{\boxdot x^{\mathsf{i}}[a]} P$$

$$\diamonddot y\{i.\, P, za'z'.\, Q, f.\, R\} \xrightarrow[\textsc{DiW}]{\diamonddot y()} \nu i f\, (P \mid R)$$

$$\diamonddot y\{i.\, P, za'z'.\, Q, f.\, R\} \xrightarrow[\textsc{DiA}]{\diamonddot y^{\mathsf{i}}(a)}$$

$$\nu i' z'\, \nu i z\, (Q^{\mathsf{i}} \mid P \mid (b \leftrightarrow b')^{b'}_{b} \otimes^{i'}_{y} \diamonddot y\{i.\, i' \leftrightarrow i, za'z'.\, Q, f.\, R\}) \qquad \textsc{BoW-DiW}\ \frac{P \xrightarrow{\boxdot x[] \mid \diamonddot y()} P'}{\nu xy\, P \xrightarrow{\tau} \nu xy\, P'}$$

$$\textsc{BoA-DiA}\ \frac{P \xrightarrow{\boxdot x^{\mathsf{i}}[a] \mid \diamonddot y^{\mathsf{i}}(a')} P'}{\nu xy\, P \xrightarrow{\tau} \nu a a'\, \nu xy\, P'}$$

Figure 6: Effects, transition rules for processes

serving request at a'. It then transitions into a complex process which is essentially as follows. We first feed the initial internal state i from P to Q^{i} as old internal state z', and get back $\Gamma, A, B \otimes S$ where S is the new internal state. Note we can construct a new coemitter with the same Q and R but some given S as the initial internal state; this is exactly what T does. We apply T to the aforementioned new S and replace it, and get $\Gamma, A, B \otimes \diamonddot^{\ominus} C$.

Finally, the communication rules BoW-DiW and BoA-DiA specify the interaction. BoW-DiW says if the emitter signals end, then the coemitter will end. BoA-DiA says if the emitter emits an effect, the coemitter will handle it.

Note that both Baelde [2012] and Lindley and Morris [2016] gave reduction semantics; the LTS semantics give here is new. A benefit of LTS is that each side of communication transitions on their own. This is particularly useful for primitive effects (whose coemitter are not expressed in the language), because such an emitter would still signal effects by BoAA, upon which the standard toolbox of bisimilarity can be applied. In comparison, reduction semantics needs both sides to form a redex. Ahman and Bauer [2020] attempted to solve it by defining in the language a top-level layer (i.e. operating system) handling primitive effects; this is simply shifting the issue, as the top-level layer would still need mechanisms to interact with human or communicate over network.

Example 1 (Linear Cell)**.** A linear cell is a simple buffer that is either empty or contains a piece of datum X. It is linear in that it becomes empty once read, and can only be written to when empty; the content is never duplicated or discarded. As a result, X could be any session type and not a positive one. However, for easier understanding, one can assume X to be a positive type (such as integer) for now. We first define:

$$S := \mathbf{1} \oplus X \qquad \Theta := \mathsf{get} : (\mathbf{1} : \bot \mathbin{\&} X^\perp), \mathsf{put} : (X : X^\perp \mathbin{\&} \bot)$$

The internal state S is a buffer, a $\oplus$ where left means ‘empty’ and right means ‘non-empty’ along with the content X. The effect environment Θ is defined from the perspective of the emitter, and has two operations. In get, the emitter will send a trivial request and it must then be prepared for two possible outcomes in the response: left means the cell is empty, while right allows for the reception of the content. In put, the emitter will send the datum to be stored, and prepare for two possible outcomes in the response: left means the cell is occupied already along with the datum bounced back, right means the datum is successfully stored.

The initialization is an arbitrary process P supplied by the user; the finalization process simply returns the buffer.

$$P \vdash \Gamma, i : S \qquad R := f \leftrightarrow y \vdash f : S^\perp, y : S$$

We then define the effect handling. First note that the coemitter views the effects dually to the emitter:

$$\Theta^\perp = \mathsf{get} : (\bot : \mathbf{1} \oplus X), \mathsf{put} : (X^\perp : X \oplus \mathbf{1})$$

The following process handles get, where we terminate the request a', output a fresh session b' for the response, and finally we do two things: we forward the old buffer z to the response b', and we set the new buffer z' to be empty.

$$Q^{\mathsf{get}} \vdash z : S^\perp, a' : \bot, z' : (\mathbf{1} \oplus X) \otimes S$$
$$Q^{\mathsf{get}} := a'().\, z'[b'].\, (z \leftrightarrow b' \mid z'[\mathrm{L}].\, z'[].\, \mathbf{0})$$

The following process handles put; it checks the old buffer z, and proceeds accordingly:

$$Q^{\mathsf{put}} := z.\mathsf{case}\{\mathrm{L}{:}Q_{\mathsf{empty}}, \mathrm{R}{:}Q_{\mathsf{full}}\} \vdash z : S^\perp, a' : X^\perp, z' : (X \oplus \mathbf{1}) \otimes S$$

If the old buffer z is empty, we end the old buffer z, output a fresh session b' for the response, and do two things: we signal success on the response b', and we forward the datum from request a' to the new buffer z'.

$$Q_{\mathsf{empty}} \vdash z : \bot, a' : X^\perp, z' : (X \oplus \mathbf{1}) \otimes S$$
$$Q_{\mathsf{empty}} := z().\, z'[b'].\, (b'[\mathrm{L}].\, b'[].\, \mathbf{0} \mid z'[\mathrm{R}].\, a' \leftrightarrow z')$$

If the old buffer z is non-empty, we simply forward the request a' to the response b', and forward the old buffer z to the new buffer z'.

$$Q_{\mathsf{full}} \vdash z : X^\perp, a' : X^\perp, z' : (X \oplus \mathbf{1}) \otimes S$$
$$Q_{\mathsf{full}} := z'[b'].\, (b'[\mathrm{R}].\, a' \leftrightarrow b' \mid z'[\mathrm{R}].\, z \leftrightarrow z')$$

Finally, we put everything together to define a linear cell:

$$\mathsf{cell}_y^X(i.\,P) := \diamonddot y\{i.\,P, za'z'.\,Q, f.\,R\} \vdash \Gamma, y : \diamonddot^{\Theta^\perp} S$$

Bisimilarity results that portrays the cell's behaviours are proved (see theorem 40 in section 3.D). Positive cells [Rocha and Caires, 2021] can be defined in a similar manner (see example 18 in section 3.D). Howerver, they are less useful with sessions and we will not discuss about them further.

Example 2 (Sequential Access)**.** We now demonstrate that our system allows multiple *sequential* accesses in a single emitter. We first specialize example 1 and obtain a linear cell of **1** starting out empty:

$$\mathsf{cell}_y^{\mathbf{1}}(i.\,i[\mathrm{L}].\,\mathbf{0}_i) \vdash y : \diamonddot^{\Theta^\perp} S$$

The emitter will get from the cell and then put to the cell, in that order. Both responses will be of type $\mathbf{1} \oplus \mathbf{1}$, which will be discarded for simplicity (recall that positive types can be discarded: given any $P \vdash \Gamma$, we have $\mathsf{del}(x).P \vdash x : (\mathbf{1} \oplus \mathbf{1})^\perp, \Gamma$).

We now define the emitter. Note that the two $\mathbf{0}_a$ have different meanings: the former is the trivial request of get, while the latter is the datum to put. Also note that in the end we forward the returning value of the coemitter (which is the buffer as defined in example 1) to s.

$$\begin{aligned}\mathsf{main} :=\; & \boxdot x^{\mathsf{get}}[a].\,(\mathbf{0}_a \mid x(b).\,\mathsf{del}(b).\boxdot x^{\mathsf{put}}[a].\,(\mathbf{0}_a \mid x(b).\,\mathsf{del}(b).\boxdot x[].\,x \leftrightarrow s)) \\ & \vdash x : \boxdot^{\Theta} S^\perp, s : S\end{aligned}$$

Finally we connect main to cell. The first effect get will fail because the cell was empty, and the second effect put will succeed, which also decides the final internal state $s : S$. Note this is the only possible outcome - the second request cannot run before the first. The process will transition as follows. The first transition signals that the cell is non-empty, and the second transition signals the trivial content.

$$\nu xy\,(\mathsf{main} \mid \mathsf{cell}_y^{\mathbf{1}}(i.\,i[\mathrm{L}].\,\mathbf{0}_i)) \xRightarrow{s[\mathrm{R}]} \xRightarrow{s[]} \mathbf{0}$$

Example 3 (Merging Coemitters)**.** Given two coemitters, we can merge them into one, so that the merged coemitter is capable of handling effects from both sub-coemitters. The trick is to use $\otimes$ on the sub-coemitters as the internal state of the merged coemitter, and simply forward effects to the corresponding sub-coemitter to get handled. Let $S := \diamonddot^{\Theta} C \otimes \diamonddot^{\Omega} D$. Given $P \vdash \Gamma, y : S$, we define the following merge process

$$\mathsf{merge}_y(i.P) := \diamonddot y\{i.\,P, za'z''.\,\vec{Q}, f.\,f \leftrightarrow y\} \vdash \Gamma, y : \diamonddot^{\Theta,\Omega} S$$

where $\vec{Q}$ has, $\mathsf{i} : (A : B) \in \Theta$,

$$\begin{aligned} Q^{\mathsf{i}}_{L} :=& z(z').\, \boxdot z'^{\mathsf{i}}[a].\, (a \leftrightarrow a' \mid z'(b).\, z''\langle b\rangle.\, z''\langle z'\rangle.\, z \leftrightarrow z'') \\ &\vdash z : S^{\perp}, a' : A, z'' : B \otimes S \end{aligned}$$

and for each $\mathsf{i} : (A : B) \in \Omega$ such that i is not in Θ,

$$\begin{aligned} Q^{\mathsf{i}}_{R} :=& z(z').\, \boxdot z^{\mathsf{i}}[a].\, (a \leftrightarrow a' \mid z(b).\, z''\langle b\rangle.\, z''\langle z\rangle.\, z' \leftrightarrow z'') \\ &\vdash z : S^{\perp}, a' : A, z'' : B \otimes S \end{aligned}$$

The the versions Q^{i}_{L} and Q^{i}_{R} differ only in the endpoint used for i and which correspond to the left and right type in $\diamonddot^{\Theta} C$ and $\diamonddot^{\Omega} D$, respectively. In case of clashes of effect names in Θ and Ω our implementation prioritises the first but one could prioritize the second by selecting substituting the corresponding element in $\vec{Q}$ (this bias will be relevant later in example 11). The resulting $\diamonddot^{\Theta,\Omega} S$ can handle both Θ and Ω effects, and returns S which is the left-over of both sub-coemitters. $C \otimes D$ can be derived from S and is in fact enough for most scenarios, but returning S is needed if one wishes to use the two sub-coemitters afterwards, as will be demonstrated later (example 9). Finally, observe that requiring a premise $P \vdash \Gamma, x : \diamonddot^{\Theta} C \mid \Delta, x' : \diamonddot^{\Omega} D$ would be less general compared to the current $P \vdash \Gamma, \Delta, x : \diamonddot^{\Theta} C \otimes \diamonddot^{\Omega} D$.

Remark 1 (Merging and Pairing). Merging is similar to pairing in Ahman and Bauer [2020]. However, pairing does not provide modularity because the definition of the paired runner requires the full internal knowledge of the sub-runners. In particular, the internal state of the paired runner is simply $C_0 \times C_1$ where C_0, C_1 are the internal states of the sub-runners. Our merging requires only external knowledge of the coemitters such as Ω, Θ, and therefore provides modularity.

By observing the derivation of merge, it appears that the sub-coemitters do not interfere with each other in the handling of effects. The following theorem formalizes this observation.

Theorem 19 (Independence of the sub-coemitters)**.** *For any*

$$P \vdash \Gamma_0, y_0 : \diamonddot^{\Theta} C \mid \Gamma_1, y_1 : \diamonddot^{\Omega} D$$

if $P \xrightarrow{\diamonddot y_0{}^{i}(a')} P'$, *then*

- $\mathsf{merge}_y(y_1.y_1[y_0].\, P) \xrightarrow{\diamonddot y^{i}(a')} Z$,
- $Z \vdash y_1' : (\diamonddot^{\Omega} D)^{\perp}, y_0' : (B \otimes \diamonddot^{\Theta} C)^{\perp}, y : B \otimes \diamonddot^{\Theta,\Omega}(\diamonddot^{\Theta} C \otimes \diamonddot^{\Omega} D)$, *and*
- $Z \approx \nu y_1 y_1' \, \nu y_0 y_0' \, (P' \mid y_0'(b').\, y\langle b'\rangle.\, \mathsf{merge}_y(i.i\langle y_0'\rangle.\, i \leftrightarrow y_1'))$.

Note that $y_0'(b').\, y\langle b'\rangle.\, \mathsf{merge}_y(i.i\langle y_0'\rangle.\, i \leftrightarrow y_1')$ is a thin wrapper around merge to allows for the extra B; the detour is needed because $\otimes$ is difficult in sequent calculus. The intuition of the theorem is that if y_0 handles effect i and merges with y_1, it will be equivalent to the process that merges y_0 with y_1 and handles effect i. The equivalence has two parts: the handlings of effect i are equivalent, and the remaining merged coemitters are equivalent.

3.4 Races

In the previous section we formulated the interaction between a single emitter and a single coemitter, which supports sequentiality (sequencing of operations). Recalling the design criteria discussed in the Introduction, we wish to allow multiple emitters to share accesses to a single coemitter. Suprisingly, as we will show in section 3.4.1, our system already supports user-defined deterministic sharing. To allow non-deterministic sharing (i.e., races), we extend our system in section 3.4.2 with a primitive with the same type as user-defined sharing but with non-deterministic semantics.

3.4.1 Deterministic Sharing

We show that users can define (deterministic) sharing of effects on their own. We start by stating some propositions, whose proofs can be found in the supplementary material. They are related to functoriality, (co)strength in monoidal categories and multiplication of (co)monads, the details of which we leave to future work.

In the following processes, the process lift 'lifts' C in or out of the returning value of the emitter and coemitter, without changing the effects. There is also an omitted right version that lifts D, as well as the special case lift* when $D = \perp$. The process flatten flattens an emitter that returns an emitter by 'compressing' all effects in one go.

Proposition 20. *We can derive the following; each row contains two implications that are equivalent.*

$$\begin{array}{rrr}
(C \multimap D) \multimap (\boxbox^{\Theta} C \multimap \boxbox^{\Theta} D) & (C \multimap D) \multimap (\Diamonddot^{\Theta} C \multimap \Diamonddot^{\Theta} D) & \text{(fmap)} \\
\Diamonddot^{\Theta}(C \parr D) \multimap (C \parr \Diamonddot^{\Theta} D) & (C \otimes \boxbox^{\Theta} D) \multimap \boxbox^{\Theta}(C \otimes D) & \text{(left lift)} \\
\boxbox^{\Theta} \boxbox^{\Theta} C \multimap \boxbox^{\Theta} C & \Diamonddot^{\Theta} C \multimap \Diamonddot^{\Theta} \Diamonddot^{\Theta} C & \text{(flatten)}
\end{array}$$

Consequently, the following process terms exist:

$$\begin{aligned}
T \vdash x : D, y : C \Rightarrow \mathsf{fmap}_{xy}(T) &\vdash x : \boxbox^{\Theta} D, y : \Diamonddot^{\Theta^{\perp}} C \\
\mathsf{lift}_{xcy} &\vdash c : C, y : \Diamonddot^{\Theta} D, x : \boxbox^{\Theta^{\perp}}(C^{\perp} \otimes D^{\perp}) \\
\mathsf{flatten}_{xy} &\vdash x : \boxbox^{\Theta} C, y : \Diamonddot^{\Theta^{\perp}} \Diamonddot^{\Theta^{\perp}} C^{\perp}
\end{aligned}$$

The following proposition is related to (op)lax monoidal functors (but again we leave out a detailed exploration of this fact to future work).

Proposition 21 (Deterministic Sharing)**.** *There are processes of type* $\boxdot^{\ominus}C \otimes \boxdot^{\ominus}D \multimap \boxdot^{\ominus}(C \otimes D)$ *or equivalently* $\diamonddot^{\ominus}(C \parr D) \multimap \diamonddot^{\ominus}C \parr \diamonddot^{\ominus}D$*.*

Proof. We give two proofs. Each of them corresponds to a process which is omitted.

$$\begin{aligned}
&\boxdot^{\ominus}\, C \otimes \boxdot^{\ominus} D \\
\multimap\ &\boxdot^{\ominus}\, (C \otimes \boxdot^{\ominus} D) && \text{(right } \mathsf{lift}\text{)} \\
\multimap\ &\boxdot^{\ominus}\, \boxdot^{\ominus}(C \otimes D) \\
& && \text{(left } \mathsf{fmap}(\mathsf{lift})\text{)} \\
\multimap\ &\boxdot^{\ominus}\, (C \otimes D) && \text{(}\mathsf{flatten}\text{)}
\end{aligned}$$

$$\begin{aligned}
&\boxdot^{\ominus}\, C \otimes \boxdot^{\ominus} D \\
\multimap\ &\boxdot^{\ominus}\, (\boxdot^{\ominus} C \otimes D) && \text{(left } \mathsf{lift}\text{)} \\
\multimap\ &\boxdot^{\ominus}\, \boxdot^{\ominus}(C \otimes D) \\
& && \text{(right } \mathsf{fmap}(\mathsf{lift})\text{)} \\
\multimap\ &\boxdot^{\ominus}\, (C \otimes D) && \text{(}\mathsf{flatten}\text{)}
\end{aligned}$$

□

The proofs carry computational meanings. In the left proof, lift first propagates all effects in $\boxdot^{\ominus}C$, then another lift propagates all effects in $\boxdot^{\ominus}D$. We can say that it represents a sharing policy that prioritizes the left subtree. The right derivation is the symmetric policy, which prioritizes the right subtree; one can also define one that alternates between left and right subtrees; etc.. Sharing given this way is deterministic: if one CUT's a policy with two parallel emitters, then the outcome is completely decided by the policy.

3.4.2 Non-deterministic Sharing

To model real world concurrency, we introduce non-deterministic sharing (i.e. races) by means of a new rule, BoC, in fig. 7, Note that BoC is derivable by proposition 21, and thus does not change the logical aspects of our system. What makes this rule special is the associated LTS semantics, given by the other three rules in fig. 7. The LTS semantics allows effects from both left and right subtrees to propagate non-deterministically. For clarify we also list the LTS projected to processes in fig. 8 and environments in fig. 3.B.2 in section 3.B.

The LTS of BoC is unlike that of BoA or BoW. Recall that $\boxdot x^{\mathsf{i}}[a].\,P$ simply signals the label $\boxdot x^{\mathsf{i}}[a]$. However, had we let $\boxdot x[x_0, x_1]\ P$ signal the label $\boxdot x[x_0, x_1]$, we would need extra LTS rules for the coemitter to react to this; we want to avoid that since the concern of sharing emitters should be kept separate from that of coemitters.

Instead, BoC is structural, similar to CUT. It does not signal labels by itself, but propagates labels from P with modifications if necessary. BoCP propagates labels unrelated to effects, which is similar to RES.

BoCA0 propagates an effect from the left emitter; the symmetric BoCA1 is omitted for space. BoCW collects two ending emitters and end. Both rules

BoC

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x_0 : \boxbox^{\Theta} C_0 \mid \Delta, x_1 : \boxbox^{\Theta} C_1}{\boxbox x[x_0, x_1]\, P \vdash \mathcal{G} \mid \Gamma, \Delta, x : \boxbox^{\Theta}(C_0 \otimes C_1)}$$

BoCP

$$\frac{\begin{array}{c} x_0, x_1 \notin \mathrm{F{\small N}}(l) \\ P \vdash \mathcal{G} \mid \Gamma, x_0 : \boxbox^{\Theta} C_0 \mid \Delta, x_1 : \boxbox^{\Theta} C_1 \xrightarrow{l} P' \vdash \mathcal{G}' \mid \Gamma', x_0 : \boxbox^{\Theta} C_0 \mid \Delta', x_1 : \boxbox^{\Theta} C_1 \end{array}}{\boxbox x[x_0, x_1]\, P \vdash \mathcal{G} \mid \Gamma, \Delta, x : \boxbox^{\Theta}(C_0 \otimes C_1) \xrightarrow{l} \boxbox x[x_0, x_1]\, P' \vdash \mathcal{G}' \mid \Gamma', \Delta', x : \boxbox^{\Theta}(C_0 \otimes C_1)}$$

BoCA0

$$\frac{\begin{array}{c} x_0, x_1 \notin \mathrm{F{\small N}}(l) \qquad \mathsf{i} : (A : B) \in \Theta \qquad P \vdash \mathcal{G} \mid \Gamma_0, \Gamma_1, x_0 : \boxbox^{\Theta} C_0 \mid \Delta, x_1 : \boxbox^{\Theta} C_1 \\ \xrightarrow{l \mid \boxbox x_0{}^{\mathsf{i}}[a]} P' \vdash \mathcal{G}' \mid \Gamma_0, a : A \mid \Gamma_1, x_0 : B \parr \boxbox^{\Theta} C_0 \mid \Delta', x_1 : \boxbox^{\Theta} C_1 \end{array}}{\begin{array}{c} \boxbox x[x_0, x_1]\, P \vdash \mathcal{G} \mid \Gamma_0, \Gamma_1, \Delta, x : \boxbox^{\Theta}(C_0 \otimes C_1) \\ \xrightarrow{l \mid \boxbox x^{\mathsf{i}}[a]} \nu x_0 y_0\, \nu x_1 y_1\, (P' \mid Q) \vdash \mathcal{G} \mid \Gamma_0, a : A \mid \Gamma_1, \Delta', x : B \parr \boxbox^{\Theta}(C_0 \otimes C_1) \end{array}}$$

where $Q := x(b).\, y_0\langle b\rangle.\, \boxbox x[x_0, x_1]\, (x_0 \leftrightarrow y_0 \mid x_1 \leftrightarrow y_1) \vdash$

$y_0 : (B \parr \boxbox^{\Theta} C_0)^{\perp}, y_1 : (\boxbox^{\Theta} C_1)^{\perp}, x : B \parr \boxbox^{\Theta}(C_0 \otimes C_1)$

BoCW

$$\frac{\begin{array}{c} x_0, x_1 \notin \mathrm{F{\small N}}(l) \\ P \vdash \mathcal{G} \mid \Gamma, x_0 : \boxbox^{\Theta} C_0 \mid \Delta, x_1 : \boxbox^{\Theta} C_1 \xrightarrow{l \mid \boxbox x_0[] \mid \boxbox x_1[]} P' \vdash \mathcal{G}' \mid \Gamma, x_0 : C_0 \mid \Delta, x_1 : C_1 \end{array}}{\boxbox x[x_0, x_1]\, P \vdash \mathcal{G} \mid \Gamma, \Delta, x : \boxbox^{\Theta}(C_0 \otimes C_1) \xrightarrow{l \mid \boxbox x[]} \nu x_0 y_0\, \nu x_1 y_1\, (P' \mid Q) \vdash \mathcal{G}' \mid \Gamma, \Delta, x : C_0 \otimes C_1}$$

where $Q := x\langle y_0\rangle.\, x \leftrightarrow y_1 \vdash y_0 : C_0^{\perp}, y_1 : C_1^{\perp}, x : C_0 \otimes C_1$

Figure 7: Races, typing rules and transition rules for derivations.

$$\textsc{BoCP}\quad \frac{x_0, x_1 \notin \textsc{Fn}(l) \qquad P \xrightarrow{l} P'}{\boxdot x[x_0, x_1]\, P \xrightarrow{l} \boxdot x[x_0, x_1]\, P'}$$

$$\textsc{BoCW}\quad \frac{x_0, x_1 \notin \textsc{Fn}(l) \qquad P \xrightarrow{l \mid \boxdot x_0[] \mid \boxdot x_1[]} P'}{\boxdot x[x_0, x_1]\, P \xrightarrow{l \mid \boxdot x[]} \nu x_0 y_0\, \nu x_1 y_1\, (P' \mid x\langle y_0\rangle.\, x \leftrightarrow y_1)}$$

$$\textsc{BoCA0}\quad \frac{x_0, x_1 \notin \textsc{Fn}(l) \qquad P \xrightarrow{l \mid \boxdot x_0{}^{\mathsf{i}}[a]} P'}{\boxdot x[x_0, x_1]\, P \xrightarrow{l \mid \boxdot x^{\mathsf{i}}[a]} \nu x_0 y_0\, \nu x_1 y_1\, (P' \mid x(b).\, y_0\langle b\rangle.\, \boxdot x[x_0, x_1]\, (x_0 \leftrightarrow y_0 \mid x_1 \leftrightarrow y_1))}$$

Figure 8: Races, transition rules for processes.

include a general l; this is to satisfy non-interference (see theorem 32). Both rules rely on some auxiliary term Q to work around syntactic limitation of sequent calculus.

Remark 2 (Parallelism and Concurrency). There are two kinds of computations in our system, managed by different mechanisms. Pure computations are given by the base system πLL [Montesi and Peressotti, 2021]. It is pure in the sense of satisfying the diamond property. In particular, it includes HMix and its semantics which allows independent processes to run simultaneously by Syn. This corresponds to parallelism. On the other hand, effectful computations are given by our novel extensions. Effectful computations are not pure in the sense of the partial diamond property (see theorem 34 in section 3.B). They include BoC which allows racing between emitters, and its semantics which decides the outcomes of races. This corresponds to concurrency.

Example 4 (Parallel Access)**.** Following example 2, we can now define two parallel emitters each trying to access the linear cell:

$$\mathsf{main}_0 := \boxdot x_0{}^{\mathsf{get}}[a].\, (\mathbf{0}_a \mid x_0(b).\, \mathsf{del}(b).\boxdot x_0[].\, \mathbf{0}_{x_0}) \vdash x_0 : \boxdot^{\Theta}\mathbf{1}$$
$$\mathsf{main}_1 := \boxdot x_1{}^{\mathsf{put}}[a].\, (\mathbf{0}_a \mid x_1(b).\, \mathsf{del}(b).\boxdot x_1[].\, \mathbf{0}_{x_1}) \vdash x_1 : \boxdot^{\Theta}\mathbf{1}$$

we apply BoC on them and fmap the return value from $\mathbf{1} \otimes \mathbf{1}$ to $\mathbf{1}$:

$$\mathsf{main} := \boxdot x[x_0, x_1]\, (\mathsf{main}_0 \mid \mathsf{main}_1) \vdash x : \boxdot^{\Theta}(\mathbf{1} \otimes \mathbf{1})$$
$$F := \mathsf{fmap}_{x'y'}(y'(y'').\, y'().\, y''().\, \mathbf{0}_{x'}) \vdash x'\ \boxdot^{\Theta}\, \mathbf{1}, y' : (\boxdot^{\Theta}(\mathbf{1} \otimes \mathbf{1}))^{\perp}$$
$$\mathsf{main'} := \nu x y'\, (F \mid \mathsf{main}) \vdash x' : \boxdot^{\Theta}\mathbf{1}$$

$$\textsc{BoR}\;\frac{Q \vdash e : E^{\perp}, x : \boxbox^{\Theta} C, z : (E \oplus D) \otimes \Diamonddot^{\Theta^{\perp}} C^{\perp} \qquad P \vdash \Gamma, d : D^{\perp}, x : \boxbox^{\Theta} C}{\boxbox x(e)\{z.\, Q\}d.\, P \vdash \Gamma, e : E^{\perp}, x : \boxbox^{\Theta} C}$$

$$\mathcal{D} := \boxbox x(e)\{z.\, Q\}d.\, P \vdash \Gamma, e : E^{\perp}, x : \boxbox^{\Theta} C \xrightarrow[\textsc{BoRR}]{\tau}$$

$$\frac{Q \vdash e : E^{\perp}, x : \boxbox^{\Theta} C, z : (E \oplus D) \otimes \Diamonddot^{\Theta^{\perp}} C^{\perp} \qquad \dfrac{\mathcal{D}[d/e] \vdash \Gamma, d : E^{\perp}, x : \boxbox^{\Theta} C \qquad P \vdash \Gamma, d : D^{\perp}, x : \boxbox^{\Theta} C}{\mathcal{E} := x(d).\, d.\mathsf{case}\{\textsc{l}{:}\mathcal{D}[d/e], \textsc{r}{:}P\} \vdash \Gamma, d : (E^{\perp} \mathbin{\&} D^{\perp}) \parr \boxbox^{\Theta} C}}{\nu dz\, (Q \mid \mathcal{E}) \vdash \Gamma, e : E^{\perp}, x : \boxbox^{\Theta} C}$$

Figure 9: Do-Until, typing rule and transition rules for derivations

On the other side, we lift the return value of the coemitter:

$$\mathsf{cell'} := \nu xy\, (\mathsf{cell}^{\mathbf{1}}_{y}(i.\, i[\textsc{l}].\, \mathbf{0}_i) \mid \mathsf{lift}{*}_{xcy'}) \vdash c : S, y' : \Diamonddot^{\Theta^{\perp}} \perp$$

Finally we can connect main' to cell' and get $\nu x'y'\,(\mathsf{main'} \mid \mathsf{cell'}) \vdash c : S$, where c signals the final state of the cell. There are two possible outcomes. In the first outcome, get fails followed by put succeeding, and the cell ends with $\mathbf{1}$. In the second outcome, put succeeds followed by get succeeding (because there is content in the cell), and the cell ends up being empty. Transitions are similar to those in example 2 and omitted.

The application of fmap and lift* makes the example somewhat cumbersome, we therefore postpone more examples until section 3.7, where we consider a higher-level language.

3.5 Synchronization

In example 4 above, the emitters were forced to continue even when get and put failed. In this section we introduce a do-until primitive, which repeatedly performs an effectful computation until a certain condition is met. This allows emitters to synchronize via shared effects. The cost is that do-until introduces livelocks — there might be an infinite series of τ transitions.

The do-until extension contains only one typing rule and one LTS rule, shown in fig. 9. For clarity we also list the rules projected to processes in fig. 10, and environments in fig. 3.B.3 in section 3.B. E is the type of the iteration variable and D is the type of the result. Process Q has a complex type: it takes the last iteration variable at e, and takes a coemitter at x

$$\mathcal{D} := \boxbox x(e)\{z.\, Q\}d.\, P \xrightarrow[\text{BoRR}]{\tau} \nu dz\, (Q \mid x(d).\, d.\mathsf{case}\{\textsc{l}{:}\mathcal{D}[d/e], \textsc{r}{:}P\})$$

Figure 10: Do-Until, transition rules for processes

(by exposing as an emitter), and produces $E \oplus D$ with $\diamonddot^{\Theta^\perp} C^\perp$ at z. The former has two options where left indicates continuing with a new iteration variable and right indicates ending with a result. The latter is the 'left-over' of the handler at x. Both the result and the left-over are passed to the continuation P.

In the LTS semantics, $\mathcal{D}$ is the derivation associated with $\boxbox x(e)\{z.\, Q\}d.\, P$ and it simply unfolds. On the RHS, we first run Q, the outcome of which is used to pick the branch to run. Left leads to the self-reference $\mathcal{D}[d/e]$, while the right leads to P. Note that the self-reference is guarded by case, which does not propagate deeper actions. Hence another recursive unfolding will only happen after the outcome of Q reacts with case.

The new do-until primitive can be used to synchronize get and put operations on a linear cell (see example 8). Here we only look at a few simple examples to get some intuition.

Example 5 (Infinite Loop)**.** The do-until primitive suffices for expressing an infinite loop: For simplicity, let $C := D := E := \mathbf{1}$. Given a loop body $Q' \vdash x : \boxbox^{\Theta}\mathbf{1}, z : \diamonddot^{\Theta^\perp}\perp$, we define

$$Q := e().\, z[c].\, (Q' \mid c[\textsc{l}].\, \mathbf{0}_c) \vdash e : \perp, x : \boxbox^{\Theta}\mathbf{1}, z : (\mathbf{1} \oplus \mathbf{1}) \otimes \diamonddot^{\Theta^\perp}\perp$$
$$P := c().\, \boxbox x[].\, \mathbf{0}_x \vdash c : \perp, x : \boxbox^{\Theta}\mathbf{1}$$
$$\mathsf{loop}(xz.Q') := \nu ee'\, (\mathbf{0}_{e'} \mid \boxbox x(e)\{z.\, Q\}c.\, P) \vdash x : \boxbox^{\Theta}\mathbf{1}$$

The use of the fixed expression $c[\textsc{l}]$ makes the loop always continue. As a result, process P, which ends the emitter, will never be executed.

Example 6 (Spawn Bomb)**.** The simplest instance of the preceding example is one where Q' is given by $x \leftrightarrow z$; this corresponds to an empty loop body. We name such a loop dumb. If we repeatedly spawn dumb, we get more and more of them, effectively creating a spawn bomb. How does spawn work? The main emitter x forks by $\boxbox x[x_0, x_1]$. The child emitter x_0 will then be used by dumb, while the child emitter x_1 is forwarded as the next loop variable and thus becomes the next main emitter.

$$\mathsf{dumb} := \mathsf{loop}(x_0 z.x_0 \leftrightarrow z) \vdash x_0 : \boxbox^{\Theta}\mathbf{1}$$
$$\mathsf{bomb} := \mathsf{loop}(xz.\boxbox x[x_0, x_1]\, (\mathsf{dumb} \mid x_1 \leftrightarrow z)) \vdash x : \boxbox^{\Theta}\mathbf{1}$$

3.6 Metatheory

CELL enjoys the same metatheoretic results that validate the design of πLL especially, erasure (theorem 22), session fidelity (theorem 23), and progress (theorem 25). The only exceptions are results that rely on the absence of livelocks and races, notably CELL does not enjoy the readiness and diamond properties. The first states that a process is ready to perform an action on at least one endpoint for each environment in its type. This property fails in CELL because of BoR which introduces busy-waiting that might result in infinite sequences of τ-transitions. The second property states that any interleaving of concurrent actions that do not share endpoints leads to the same result. This property fails in CELL because of non-deterministic sharing and the resolution of races by BoCA0 and BoCA1. However, a similar property holds under a restricted usage of these rules (lemma 27, theorem 34).

The transition systems of processes and environments we derived from lts of derivations of CELL are coherent: the semantics of well-typed processes does not depend from runtime information about their typing (erasure) and is simulated by the semantics of types (session fidelity).

Theorem 22 (Erasure)**.** *For any derivation $\mathcal{D}$ and label l:*

- *if $\mathcal{D} \xrightarrow{l} \mathcal{D}'$, then $proc(\mathcal{D}) \xrightarrow{l} proc(\mathcal{D}')$;*
- *if $proc(\mathcal{D}) \xrightarrow{l} P'$, then $\mathcal{D} \xrightarrow{l} \mathcal{D}'$ for some $proc(\mathcal{D}') = P'$.*

Theorem 23 (Session Fidelity)**.** *If $P \vdash \mathcal{G}$ and $P \xrightarrow{l} P'$, then $P' \vdash \mathcal{G}'$ and $\mathcal{G} \xrightarrow{l} \mathcal{G}'$ for some $\mathcal{G}'$.*

It follows from theorem 22 that well-typed processes stay well-typed under any transition.

Corollary 24 (Typability Preservation)**.** *If P is well-typed and $P \xrightarrow{l} P'$, then P' is well-typed.*

Processes that are typed under non-empty hyperenvironments are not stuck.

Theorem 25 (Progress)**.** *If $P \vdash \mathcal{G}$ and $\mathcal{G} \neq \emptyset$, then $P \rightarrow$.*

Bisimilarity and strong bisimilarity are congruences for the lts of processes, so they allow for local reasoning. In fact, for any context $C[-]$, $P \approx Q$ implies that $C[P] \approx C[Q]$ and likewise for $\sim$.

Theorem 26 (Congruence)**.** *$\sim$ and $\approx$ are congruences.*

We call a transition derived without relying on BoCA0 or BoCA1 *pure* and we denote pure transitions by $\underset{=}{\rightarrow}$. Pure τ-transitions do not affect the observable behaviour of processes and derivations.

Lemma 27. *If* $\mathcal{D} \underset{=}{\xrightarrow{\tau}} \mathcal{E}$*, then* $\mathcal{D} \approx \mathcal{E}$*.*

3.7 Concurrent Effectful GV

Similar to other languages [Montesi and Peressotti, 2021, Kokke et al., 2019a] based on CP [Wadler, 2014], CELL can be cumbersome to write and read. Wadler [2014] remedies this issue by introducing GV, a functional programming language with session types, with thin translations to CP. Here we extend GV with effects; the new language is called CEGV and translates to CELL.

Terms The terms of CEGV are inductively generated by the following grammar. We omit selection over sessions ($\oplus, \&$ as in GV) because they are not used in our examples.

$$\begin{aligned}
P, Q, R, \cdots \;:=\;& \lambda x.\, P \mid P\ Q \mid \star \mid (P, Q) \mid \mathsf{let}\ (x, y) \;=\; P\ \mathsf{in}\ Q \\
&\hspace{12em}\text{(functions, units, product)} \\
& \mathsf{inl}\ P \mid \mathsf{inr}\ P \mid \mathsf{match}\ P\ \mathsf{with}\ x.\{Q_0, Q_1\} \qquad \text{(coproduct)} \\
& \mathsf{send}\ P\ Q \mid \mathsf{recv}\ P \mid \mathsf{terminate}\ P \\
&\hspace{6em}\text{(send and receive over and terminate } P\text{)} \\
& \mathsf{handler}\{i.P, a'z'.Q, y.R\} \qquad \text{(handler construction)} \\
& \mathsf{runner}\{P, za'.Q, f.R\} \qquad \text{(runner construction)} \\
& \mathsf{return}(P) \mid x \leftarrow P;\ Q \qquad \text{(monadic interface)} \\
& \mathsf{emit}^{\mathsf{i}}(P) \qquad \text{(emit effect named by } \mathsf{i} \text{ with request } P\text{)} \\
& \mathsf{doUntil}\ e.P \\
&\hspace{4em}\text{(repeat effectful } P \text{ until some condition are met)} \\
& \mathsf{using}\ P(Q) \qquad \text{(use the coemitter in } P \text{ and run } Q\text{)}
\end{aligned}$$

Effect Environment We lift the concept of effect environments from CELL to CEGV, with the catch that request and response types translate to opposite variances. This simplifies our rules, and the intuition is that from an emitter's perspective, A is outgoing request and B is incoming response.

$$\Theta, \Omega, \cdots \;:=\; \bullet \mid \Theta, \mathsf{i} : (A : B) \quad [\![\bullet]\!] \;:=\; \bullet \quad [\![\Theta, \mathsf{i} : (A : B)]\!] \;:=\; [\![\Theta]\!], \mathsf{i} : ([\![A]\!] : [\![B]\!]^{\perp})$$

Types Following Wadler [2014], we adopt the call-by-value translation from intuitionistic logic to linear logic [Girard, 1987a]. We define CEGV types and their translations to CELL types:

$$\begin{aligned}
A, B, C, D, \cdots \; := \quad & C \multimap D \mid C \to D \mid C + D \mid C \otimes D \mid \mathbf{1} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{(functional types)} \\
& \mid \, !C.D \mid \, ?C.D \\
& \text{(output/input value of type } C\text{, then behave as } D\text{)} \\
& \mid \mathsf{end}_? \mid \mathsf{end}_! \qquad\qquad\qquad \text{(end-of-session)} \\
& \mid \boxdot^{\Theta} C \mid \Diamonddot^{\Theta} C \qquad\qquad \text{(emitter and handler)}
\end{aligned}$$

$$[\![C \multimap D]\!] := [\![C]\!]^{\perp} \parr [\![D]\!] \qquad [\![C \to D]\!] := !([\![C]\!]^{\perp} \parr [\![D]\!])$$

$$[\![C + D]\!] := [\![C]\!] \oplus [\![D]\!] \qquad [\![C \otimes D]\!] := [\![C]\!] \otimes [\![D]\!] \qquad [\![\mathbf{1}]\!] := \mathbf{1}$$

$$[\![!C.D]\!] := [\![C]\!]^{\perp} \parr [\![D]\!] \qquad [\![?C.D]\!] := [\![C]\!] \otimes [\![D]\!] \qquad [\![\mathsf{end}_!]\!] := \perp$$

$$[\![\mathsf{end}_?]\!] := \mathbf{1} \qquad [\![\boxdot^{\Theta} C]\!] := \boxdot^{[\![\Theta]\!]}[\![C]\!] \qquad [\![\Diamonddot^{\Theta} C]\!] := \Diamonddot^{[\![\Theta]\!]^{\perp}}[\![C]\!]$$

Note that the translations of $C \multimap D$ and $!C.D$ (and other pairs of connectives) collide in CELL; we still keep them separated in CEGV, just to avoid confusion. Also note that Θ is negated in $[\![\Diamonddot^{\Theta} C]\!]$ — because in CEGV we always view effects from the perspective of the emitter, in order to be more comparable to existing effect systems.

Duality Session types C_S is a subset of types characterized by duality $\overline{C_S}$, inductively defined as

$$\overline{!D.C_S} := ?D.\overline{C_S} \qquad \overline{?D.C_S} := !D.\overline{C_S} \qquad \overline{\mathsf{end}_!} := \mathsf{end}_? \qquad \overline{\mathsf{end}_?} := \mathsf{end}_!$$

$$\overline{\boxdot^{\Theta} C_S} := \Diamonddot^{\Theta}\overline{C_S} \qquad \overline{\Diamonddot^{\Theta} C_S} := \boxdot^{\Theta}\overline{C_S}$$

We have a finer notion of session type than Wadler [2014]. For example, their system allows $!D.C$ only if C is a session type. Instead, our system allows $!D.C$ in general, but considers it as a session type when C is a session type. Similarly, $\boxdot^{\Theta} C_S$ is a session type and thus enjoys some flexibility as we will see, while $\boxdot^{\Theta} C$ in general is still comparable to usual effect systems. Also note that Θ is not negated when negating $\boxdot$ and $\Diamonddot$, which echos the point above that effect environments are always specified from the emitter's perspective to be more comparable to existing effect systems.

Unlimited Types Unlimited types C_U is a subset of types that enjoys weakening and contraction, inductively defined as:

$$C_U, D_U, \cdots := \mathbf{1} \mid C \to D \mid C_U + D_U \mid C_U \otimes D_U$$

$$\textsc{Send}\ \frac{\Gamma \vdash P : C \qquad \Delta \vdash Q : !C.D}{\Gamma, \Delta \vdash \mathsf{send}\ P\ Q : D} \qquad \textsc{Recv}\ \frac{\Gamma \vdash P : ?C.D}{\Gamma \vdash \mathsf{recv}\ P : C \otimes D} \qquad \textsc{Term}\ \frac{\Gamma \vdash P : \mathsf{end}_?}{\Gamma \vdash \mathsf{terminate}\ P : \mathbf{1}}$$

$$\textsc{Pop}\ \frac{\Gamma, x : C_S \vdash P : \mathsf{end}_!}{\Gamma \vdash \mathsf{pop}_x(P) : \overline{C_S}} \qquad \textsc{Push}\ \frac{\Gamma \vdash P : C_S}{\Gamma, x : \overline{C_S} \vdash \mathsf{push}_x(P) : \mathsf{end}_!}$$

Figure 11: CEGV, typing rules of sessions

Pure Computation Following Wadler [2014], a pure computation translates to a process with a designated port for returning a value:

$$[\![\Gamma \vdash P : D]\!]_z := [\![P]\!]_z \vdash [\![\Gamma]\!]^\perp, z : [\![D]\!]$$

The basic type system follows Wadler [2014]. The functional fragment is omitted. The session fragment is in fig. 11. Send sends data P over session Q and returns the rest of the session. Recv receives data over session P and returns the data as well as the rest of the session. Pop and Push say that consuming a session is equivalent to producing the dual session. They allow us to connect dual processes by regular function applicaiton, and replace the connect construct in Wadler [2014].

Effectful Computation The typing judgment for effectful computations has the form $\Gamma \mid \Theta \vdash P : D$. Here P is an effectful computation returning D using values from Γ and effects from Θ. The translation to CELL is shown in the display below and yields a process $[\![P]\!]_{xzC}$, which takes a coemitter at x (by exposing as an emitter at x) and returns at z some D as well as the 'left-over' of the coemitter. Note that it is parametric on C.

$$[\![\Gamma \mid \Theta \vdash P : D]\!]_{xzC} := [\![P]\!]_{xzC} \vdash [\![\Gamma]\!]^\perp, x : \boxbox^{[\![\Theta]\!]} C, z : [\![D]\!] \otimes \Diamonddot^{[\![\Theta]\!]^\perp} C^\perp$$

The typing rules for effectful computations and their translations are given in fig. 12. Return and Bind give the usual monadic interface (proved by proposition 28). Emit emits effects. Spawn spawns P and immediately gives $\mathbf{1}$; note that P will race for effects with what follows after the Spawn clause. The translation in Spawn looks a bit complicated – the core part is $\boxbox x[x_0, x_1]$ and $x_1 \leftrightarrow z$, and uses the same trick as in example 6; the rest is just to adapt types. LetE binds pure computations (and thus session communications) in effectful computations; DoUntil repeatedly runs effectful Q until it gives result D. RelaxE is a weakening rule for unused effects.

Handlers and Runners To handle effectful computations, we provide two paradigms, corresponding to handlers [Pretnar and Plotkin, 2013] and

$$\left\llbracket \begin{array}{l}\textsc{Return}\\ \dfrac{\Gamma \vdash P : D}{\Gamma \mid \bullet \vdash \mathsf{return}(P) : D}\end{array} \right\rrbracket_{xyC} :=$$

$$\frac{\llbracket P \rrbracket_z \vdash \llbracket \Gamma \rrbracket^\perp, z : \llbracket D \rrbracket \qquad x \leftrightarrow y \vdash x : \boxdot^\bullet C, y : \Diamonddot^\bullet C^\perp}{y[z].\,(\llbracket P \rrbracket_z \mid x \leftrightarrow y) \vdash \llbracket \Gamma \rrbracket^\perp, x : \boxdot^\bullet C, y : \llbracket D \rrbracket \otimes \Diamonddot^\bullet C^\perp}$$

$$\left\llbracket \begin{array}{l}\textsc{Bind}\\ \dfrac{\Gamma \mid \Theta \vdash P : A \qquad \Delta, a : A \mid \Theta \vdash Q : B}{\Gamma, \Delta \mid \Theta \vdash a \leftarrow P; Q : B}\end{array} \right\rrbracket_{x_0 z_1 C} :=$$

$$\frac{\begin{array}{c}\llbracket P \rrbracket_{x_0 z_0 C} \vdash \llbracket \Gamma \rrbracket^\perp, x_0 : \boxdot^{\llbracket \Theta \rrbracket} C, z_0 : \llbracket A \rrbracket \otimes \Diamonddot^{\llbracket \Theta \rrbracket^\perp} C^\perp\\ \llbracket Q \rrbracket_{x_1 z_1 C} \vdash \llbracket \Delta \rrbracket^\perp, a : \llbracket A \rrbracket^\perp, x_1 : \boxdot^{\llbracket \Theta \rrbracket} C, z_1 : \llbracket B \rrbracket \otimes \Diamonddot^{\llbracket \Theta \rrbracket^\perp} C^\perp\end{array}}{\nu z_0 x_1\,(\llbracket P \rrbracket_{x_0 z_0 C} \mid x_1(a).\,\llbracket Q \rrbracket_{x_1 z_1 C}) \vdash \llbracket \Gamma \rrbracket^\perp, \llbracket \Delta \rrbracket^\perp, x_0 : \boxdot^{\llbracket \Theta \rrbracket} C, z_1 : \llbracket B \rrbracket \otimes \Diamonddot^{\llbracket \Theta \rrbracket^\perp} C^\perp}$$

$$\left\llbracket \begin{array}{l}\textsc{Emit}\\ \dfrac{\Gamma \vdash P : A \qquad \mathsf{i} : (A : B) \in \Theta}{\Gamma \mid \Theta \vdash \mathsf{emit}^{\mathsf{i}}(P) : B}\end{array} \right\rrbracket_{xzC} :=$$

$$\frac{\llbracket P \rrbracket_a \vdash \llbracket \Gamma \rrbracket^\perp, a : \llbracket A \rrbracket \qquad \overline{x \leftrightarrow z \vdash x : \llbracket B \rrbracket^\perp \parr \boxdot^{\llbracket \Theta \rrbracket} C, z : \llbracket B \rrbracket \otimes \Diamonddot^{\llbracket \Theta \rrbracket^\perp} C^\perp}}{\boxdot x^{\mathsf{i}}[a].\,(\llbracket P \rrbracket_a \mid x \leftrightarrow z) \vdash \llbracket \Gamma \rrbracket^\perp, x : \boxdot^{\llbracket \Theta \rrbracket} C, z : \llbracket B \rrbracket \otimes \Diamonddot^{\llbracket \Theta \rrbracket^\perp} C^\perp}$$

$$\left\llbracket \begin{array}{l}\textsc{Spawn}\\ \dfrac{\Gamma \vdash P : \boxdot^{\Theta} \mathbf{1}}{\Gamma \mid \Theta \vdash \mathsf{spawn}(P) : \mathbf{1}}\end{array} \right\rrbracket_{x' z C} :=$$

$$\frac{\begin{array}{c}\llbracket P \rrbracket_{x_0} \vdash \llbracket \Gamma \rrbracket^\perp, x_0 : \boxdot^{\llbracket \Theta \rrbracket} \mathbf{1} \qquad x_1 \leftrightarrow z \vdash x_1 : \boxdot^{\llbracket \Theta \rrbracket} C, z : \Diamonddot^{\llbracket \Theta \rrbracket^\perp} C^\perp\\ Q := \mathsf{fmap}_{x'y}(y(y').\,y'().\,y \leftrightarrow x') \vdash y : (\boxdot^{\llbracket \Theta \rrbracket}(\mathbf{1} \otimes C))^\perp, x' : \boxdot^{\llbracket \Theta \rrbracket} C\end{array}}{z[z'].\,\nu xy\, \boxdot x[x_0, x_1]\,(\llbracket P \rrbracket_{x_0} \mid x_1 \leftrightarrow z \mid Q \mid \mathbf{0}_{z'}) \vdash \llbracket \Gamma \rrbracket^\perp, x' : \boxdot^{\llbracket \Theta \rrbracket} C, z : \mathbf{1} \otimes \Diamonddot^{\llbracket \Theta \rrbracket^\perp} C^\perp}$$

$$\begin{array}{l}\textsc{LetE}\\ \dfrac{\Gamma \vdash P : D \qquad \Delta, d : D \mid \Theta \vdash Q : E}{\Gamma, \Delta \mid \Theta \vdash \mathsf{let}\ d\ =\ P\ \mathsf{in}\ \ Q : E}\end{array} \qquad \begin{array}{l}\textsc{DoUntil}\\ \dfrac{e : E \mid \Theta \vdash Q : E + D}{e : E \mid \Theta \vdash \mathsf{doUntil}\ e.Q : D}\end{array}$$

$$\begin{array}{l}\textsc{RelaxE}\\ \dfrac{\Gamma \mid \Theta \vdash P : D}{\Gamma \mid \Theta, \mathsf{i} : (A : B) \vdash P : D}\end{array}$$

Figure 12: CEGV, typing rules of effectful computation

$$\left\llbracket \begin{array}{c}\textsc{Inst}\\ \dfrac{\Gamma \mid \Theta \vdash P : D}{\Gamma \vdash \mathsf{inst}(P) : (D \multimap \boxdot^{\Theta} C) \multimap \boxdot^{\Theta} C}\end{array}\right\rrbracket_x :=$$

$$\dfrac{\llbracket P\rrbracket_{xz\llbracket C\rrbracket} \vdash \llbracket\Gamma\rrbracket^\perp, x : \boxdot^{\llbracket\Theta\rrbracket}\llbracket C\rrbracket, z : \llbracket D\rrbracket \otimes \diamonddot^{\llbracket\Theta\rrbracket^\perp}\llbracket C\rrbracket^\perp}{x(z).\, \llbracket P\rrbracket_{xz\llbracket C\rrbracket} \vdash \llbracket\Gamma\rrbracket^\perp, x : (\llbracket D\rrbracket \otimes \diamonddot^{\llbracket\Theta\rrbracket^\perp}\llbracket C\rrbracket^\perp) \parr \boxdot^{\llbracket\Theta\rrbracket}\llbracket C\rrbracket}$$

$$\left\llbracket \begin{array}{c}\textsc{Unit}\\ \dfrac{}{\vdash \mathsf{unit} : D \multimap \boxdot^{\Theta} D}\end{array}\right\rrbracket_z := \dfrac{\boxdot z[].\, d \leftrightarrow z \vdash d : \llbracket D\rrbracket^\perp, z : \boxdot^{\llbracket\Theta\rrbracket}\llbracket D\rrbracket}{z(d).\, \boxdot z[].\, d \leftrightarrow z \vdash z : \llbracket D\rrbracket^\perp \parr \boxdot^{\llbracket\Theta\rrbracket}\llbracket D\rrbracket}$$

$$\left\llbracket \begin{array}{c}\textsc{Handler}\\ \dfrac{\Gamma, i : S \vdash P : C \qquad \forall \mathsf{i} : (A : B) \in \Theta, a' : A, z' : B \multimap S \vdash Q : S \qquad y : D \vdash R : S}{\Gamma \vdash \mathsf{handler}\{i.P, a'z'.Q, y.R\} : \boxdot^{\Theta} D \multimap C}\end{array}\right\rrbracket_c :=$$

$$\dfrac{\dfrac{\begin{array}{c}\llbracket P\rrbracket_c \vdash \llbracket\Gamma\rrbracket^\perp, i : \llbracket S\rrbracket^\perp, c : \llbracket C\rrbracket\\ \forall \mathsf{i} : (\llbracket A\rrbracket^\perp : \llbracket B\rrbracket) \in \llbracket\Theta\rrbracket^\perp, \llbracket Q\rrbracket_z \vdash z : \llbracket S\rrbracket, a' : \llbracket A\rrbracket^\perp, z' : \llbracket B\rrbracket \otimes \llbracket S\rrbracket^\perp\\ \llbracket R\rrbracket_f \vdash f : \llbracket S\rrbracket, y : \llbracket D\rrbracket^\perp\end{array}}{\diamonddot y\{i.\, \llbracket P\rrbracket_c, za'z'.\, \llbracket Q\rrbracket_z, f.\, \llbracket R\rrbracket_f\} \vdash \llbracket\Gamma\rrbracket^\perp, y : \diamonddot^{\llbracket\Theta\rrbracket^\perp}\llbracket D\rrbracket^\perp, c : \llbracket C\rrbracket}\textsc{Di}}{c(y). \cdots \vdash \llbracket\Gamma\rrbracket^\perp, c : \diamonddot^{\llbracket\Theta\rrbracket^\perp}\llbracket D\rrbracket^\perp \parr \llbracket C\rrbracket}\textsc{Par}$$

$$\begin{array}{c}\textsc{Close}\\ \dfrac{\Gamma \mid \Theta \vdash P : D}{\Gamma \vdash \mathsf{close}(P) : \boxdot^{\Theta} D}\end{array} \qquad \begin{array}{c}\textsc{Open}\\ \dfrac{\Gamma \vdash P : \boxdot^{\Theta} D}{\Gamma \mid \Theta \vdash \mathsf{open}(P) : D}\end{array}$$

Figure 13: CEGV, typing rules of handler paradigm

$$\left\llbracket \dfrac{\Gamma \mid \Theta \vdash P : D}{\Gamma \vdash \mathsf{coinst}(P) : \Diamonddot^{\Theta} C \multimap D \otimes \Diamonddot^{\Theta} C}\,\textsc{CoInst} \right\rrbracket_z :=$$

$$\dfrac{\llbracket P \rrbracket_{xz\llbracket C \rrbracket^\perp} \vdash \llbracket \Gamma \rrbracket^\perp, x : \boxdot^{\llbracket \Theta \rrbracket} \llbracket C \rrbracket^\perp, z : \llbracket D \rrbracket \otimes \Diamonddot^{\llbracket \Theta \rrbracket^\perp} \llbracket C \rrbracket}{z(x).\, \llbracket P \rrbracket_{xz\llbracket C \rrbracket^\perp} \vdash \llbracket \Gamma \rrbracket^\perp, z : \boxdot^{\llbracket \Theta \rrbracket} \llbracket C \rrbracket^\perp \parr (\llbracket D \rrbracket \otimes \Diamonddot^{\llbracket \Theta \rrbracket^\perp} \llbracket C \rrbracket)}$$

$$\left\llbracket \dfrac{}{\vdash \mathsf{counit} : \Diamonddot^{\Theta} D \multimap D}\,\textsc{CoUnit} \right\rrbracket_z := \dfrac{\boxdot d[].\, d \leftrightarrow z \vdash d : \boxdot^{\llbracket \Theta \rrbracket} \llbracket D \rrbracket^\perp, z : \llbracket D \rrbracket}{z(d).\, \boxdot d[].\, d \leftrightarrow z \vdash z : \boxdot^{\llbracket \Theta \rrbracket} \llbracket D \rrbracket^\perp \parr \llbracket D \rrbracket}$$

$$\left\llbracket \dfrac{\Gamma \vdash P : S \qquad \forall \mathsf{i} : (A : B) \in \Theta, z : S, a' : A \vdash Q : B \otimes S \qquad f : S \vdash R : D}{\Gamma \vdash \mathsf{runner}\{P, za'.Q, f.R\} : \Diamonddot^{\Theta} D}\,\textsc{Runner} \right\rrbracket_y :=$$

$$\dfrac{\begin{array}{c}\llbracket P \rrbracket_i \vdash \llbracket \Gamma \rrbracket^\perp, i : \llbracket S \rrbracket \\ \forall \mathsf{i} : (\llbracket A \rrbracket^\perp : \llbracket B \rrbracket) \in \llbracket \Theta \rrbracket^\perp, \llbracket Q \rrbracket_{z'} \vdash z : \llbracket S \rrbracket^\perp, a' : \llbracket A \rrbracket^\perp, z' : \llbracket B \rrbracket \otimes \llbracket S \rrbracket \\ \llbracket R \rrbracket_y \vdash f : \llbracket S \rrbracket^\perp, y : \llbracket D \rrbracket\end{array}}{\Diamonddot y\{i.\, \llbracket P \rrbracket_i, za'z'.\, \llbracket Q \rrbracket_{z'}, f.\, \llbracket R \rrbracket_y\} \vdash \llbracket \Gamma \rrbracket^\perp, y : \Diamonddot^{\llbracket \Theta \rrbracket^\perp} \llbracket D \rrbracket}\,\textsc{Di}$$

Figure 14: CEGV, typing rules of runner paradigm

$$\left\llbracket \dfrac{\Gamma \mid \Omega \vdash P : \Diamonddot^{\Theta} E \qquad \Delta \mid \Theta, \Omega \vdash Q : D}{\Gamma, \Delta \mid \Omega \vdash \mathsf{using}\ P(Q) : D \otimes \Diamonddot^{\Theta} E}\,\textsc{Using} \right\rrbracket_{xwC} :=$$

$$\dfrac{\mathcal{D} \qquad \llbracket Q \rrbracket_{x'z'S^\perp} \vdash \llbracket \Delta \rrbracket^\perp, x' : \boxdot^{\llbracket \Theta, \Omega \rrbracket} S^\perp, z' : \llbracket D \rrbracket \otimes \Diamonddot^{\llbracket \Theta, \Omega \rrbracket^\perp} S \qquad \mathcal{E}}{\nu w'z'\, \nu x'y\, (\mathcal{D} \mid \llbracket Q \rrbracket_{x'z'S^\perp} \mid \mathcal{E}) \vdash \llbracket \Gamma \rrbracket^\perp, \llbracket \Delta \rrbracket^\perp, x : \boxdot^{\llbracket \Omega \rrbracket} C, w : \llbracket D \rrbracket \otimes S}$$

where

$$S := \Diamonddot^{\llbracket \Theta \rrbracket^\perp} \llbracket E \rrbracket \otimes \Diamonddot^{\llbracket \Omega \rrbracket^\perp} C^\perp \qquad \mathcal{D} := \dfrac{\llbracket P \rrbracket_{xiC} \vdash \llbracket \Gamma \rrbracket^\perp, x : \boxdot^{\llbracket \Omega \rrbracket} C, i : S}{\mathsf{merge}_y(i.\llbracket P \rrbracket_{xiC}) \vdash \llbracket \Gamma \rrbracket^\perp, x : \boxdot^{\llbracket \Omega \rrbracket} C, y : \Diamonddot^{\llbracket \Theta, \Omega \rrbracket^\perp} S}$$

$$\mathcal{E} := \dfrac{\boxdot w'[].\, w \leftrightarrow w' \vdash w' : \boxdot^{\llbracket \Theta, \Omega \rrbracket} S^\perp, w : S}{w'(d').\, w\langle d' \rangle.\, \boxdot w'[].\, w \leftrightarrow w' \vdash w' : \llbracket D \rrbracket^\perp \parr \boxdot^{\llbracket \Theta, \Omega \rrbracket} S^\perp, w : \llbracket D \rrbracket \otimes S}$$

Figure 15: CEGV, typing rule of dynamic coemitter

runners [Uustalu, 2015]. The two paradigms are well-known to be dual [Plotkin and Power, 2008, Power and Shkaravska, 2004], and the duality is made apparent in our system: the handler paradigm is about emitters, while the runner paradigm is about coemitters.

In the handler paradigm (fig. 13), Inst instantiates an effectful computation P into a function which takes a continuation and returns an emitter that runs P and the continuation in sequence. An empty emitter is given by Unit. Handler constructs a function that handles an emitter. Close converts an effectful computation to an emitter, and Open does the other direction.

In the runner paradigm (fig. 14), CoInst instantiates an effectful computation into a function that takes a coemitter and returns a result along with the left-over of the coemitter. CoUnit terminates a coemitter. Runner constructs a coemitter. Runner counterparts of Open and Close are absent, because effectful computation is inherently closer to emitter.

To be more comparable with existing effect systems, we have carefully designed both paradigms so that no session type is involved. If some of the general types are session types, however, then the distinction will be blurred in light of the Push and Pop rules, similar to the fact that they collide in classical linear logic [Hasegawa, 2002, §8].

Note that in Handler, the continuation z' is linear, meaning it cannot be discarded or duplicated. As a result, we cannot model control effects such as exceptions or non-determinism. A weaker form of exception (which we call linear exception) is still possible (See example 13 in section 3.D). We discuss this point more in section 3.8.

Dynamic coemitter Using (fig. 15) allows us to use dynamically allocated coemitters. If P gives a Θ-coemitter using Ω-effects, we can add Θ to Ω and run Q. The whole term returns the result of Q with the left-over of the Θ-coemitter, while only using Ω-effects. The left-over is retained because merge returns S instead of just $[\![E]\!] \otimes C^{\perp}$. At its core is the merge (example 3) process, which also guarantees (theorem 19) that Θ and Ω will not interfere with each other in Q.

In case of effect name clashes, Θ trumps Ω, because of how merge is defined; correspondingly, effect environment merging Θ, Ω is defined to prioritize Θ. Note that Using matches the runner paradigm because the concept of 'state' is inherently closer to coemitter.

Bisimilarity We extend bisimilarity to CEGV programs. We write $\Gamma \vdash P \approx Q : D$ if $[\![P]\!]_z \approx [\![Q]\!]_z$ and we write $\Gamma \mid \Theta \vdash P \approx Q : D$ if $[\![P]\!]_{xzC} \approx [\![Q]\!]_{xzC}$.

Proposition 28 (Monadic laws)**.** *BIND and RETURN give a monad in the sense that*

$$d \leftarrow \mathit{return}(P);\ Q \approx \mathit{let}\ d\ =\ P\ \mathit{in}\ Q \qquad \text{(left identity)}$$
$$d \leftarrow P;\ \mathit{return}(d) \approx P \qquad \text{(right identity)}$$
$$d \leftarrow P;\ (e \leftarrow Q;\ R) \approx e \leftarrow (d \leftarrow P;\ Q); R \qquad \text{(associativity)}$$

Example 7 (Translating effects)**.** A coemitter might emit further effects when handling effects. Here we consider buffered writes of X, where a program can append a value to the buffer, commit what is already in the buffer, or rollback the buffer and get back what is in the buffer.

$$\Theta := \mathsf{append} : (X : \mathbf{1}), \mathsf{commit} : (\mathbf{1} : \mathbf{1}), \mathsf{rollback} : (\mathbf{1} : [X])$$

We assume the list type ($[X]$), the empty list ($[]$) and list concatenation ($::$), which are all definable (See example 15). The coemitter of these effects will maintain a buffer $[X]$ and emit effects for actual writing, which we assume is a primitive effect and which will not be dealt with further in our example.

$$\Omega := \mathsf{write} : ([X] : \mathbf{1})$$

We now define the Θ-coemitter. The trick is to include the Ω-coemitter in the internal state of the Θ-coemitter. We can take $S := \diamondsuit^{\Omega} E \otimes [X]$ if we use RUNNER; or, dually, take $S' := [X] \multimap \boxdot^{\Omega} F$ if we use HANDLER. We will use RUNNER for now. We first define:

$$z : S, a' : X \vdash Q^{\mathsf{append}} := (\star, \mathsf{let}\ (y, l)\ =\ z\ \mathsf{in}\ \ (y, a' :: l)) : \mathbf{1} \otimes S$$
$$z : S, z' : \mathbf{1} \vdash Q^{\mathsf{commit}} :=$$
$$(\star, (\mathsf{let}\ (y, l)\ =\ z\ \mathsf{in}\ \ \mathsf{let}\ (\_, y')\ =\ \mathsf{coinst}(\mathsf{emit}^{\mathsf{write}}(l))(y)\ \mathsf{in}\ \ y', []))$$
$$: \mathbf{1} \otimes S$$
$$z : S, z' : \mathbf{1} \vdash Q^{\mathsf{rollback}} := \mathsf{let}\ (y, l)\ =\ z\ \mathsf{in}\ \ (l, (y, [])) : [X] \otimes S$$

Now we can define the coemitter. For simplicity, we let the Θ-coemitter return its internal state S.

$$y : \diamondsuit^{\Omega} E \vdash L := \mathsf{runner}\{(y, []), za'.Q, f.f\} : \diamondsuit^{\Theta} S$$

The above converts Ω-coemitters to Θ-coemitters. Had we used HANDLER, we would have derived $\boxdot^{\Theta} S' \multimap \boxdot^{\Omega} F$, which converts Θ-emitters to Ω-emitters. We would pick one of them depending on whether we have $\diamondsuit^{\Omega} E$ or $\boxdot^{\Theta} S'$. If $E = \overline{F}$, the distinction blurs with PUSH and POP.

Example 8 (Linear Cell in GV)**.** Linear cells can be defined in CEGV, but we simply translate them to the existing definitions (example 1) in CELL for simplicity. For any X we have

$$\Theta := \mathsf{put} : (X : X + \mathbf{1}), \mathsf{get} : (\mathbf{1} : \mathbf{1} + X) \qquad \mathsf{Cell}^X := \diamondsuit^{\Theta}(\mathbf{1} + X)$$

$$\left[\!\!\left[\frac{\Gamma \vdash P : \mathbf{1} + X}{\Gamma \vdash \mathsf{cell}^X(P) : \mathsf{Cell}^X}\right]\!\!\right]_y := \frac{[\![P]\!]_i \vdash [\![\Gamma]\!]^\perp, i : \mathbf{1} \oplus [\![X]\!]}{\mathsf{cell}^X_y(i.\, [\![P]\!]_i) \vdash [\![\Gamma]\!]^\perp, y : \diamondsuit^{[\![\Theta]\!]^\perp}(\mathbf{1} \oplus [\![X]\!])}$$

On the other side, we define synchronized get and put using DoUntil.

$$\mid \mathsf{get} : (\mathbf{1} : \mathbf{1} + X) \vdash \mathsf{sget} := \mathsf{let}\ e \ = \ \star\ \mathsf{in}\ \ \mathsf{doUntil}\ e.\mathsf{emit}^{\mathsf{get}}(\star) : X$$
$$e : X \mid \mathsf{put} : (X : X + \mathbf{1}) \vdash \mathsf{sput}_e := \mathsf{doUntil}\ e.\mathsf{emit}^{\mathsf{put}}(e) : \mathbf{1}$$

Example 9 (Escaping Instances)**.** Using example 8, we can write a program that uses linear cells in a rich way. We assume natural numbers $\mathbb{N}$ with literal constants and addition. Intuitively we should have $M \approx \mathsf{return}((15, cell^{\mathbb{N}}(\mathsf{inr}\ 10)))$, where the second element is the left-over cell containing 10. The cell seems to have 'escaped' the scope where it is allocated and used, which is fine because Cell^X is precisely the coemitter of the cell and thus there is no danger for un-handled effects. Indeed, one can invoke Using on the returned $\mathsf{Cell}^{\mathbb{N}}$ to access the 10 stored within. Note that M has empty effect environment, because the uses of sget and sput are encapsulated by Using.

$$\begin{aligned}
&\mid\ \vdash M : \mathbb{N} \otimes \mathsf{Cell}^{\mathbb{N}} \\
&M := \mathsf{let}\ y \ = \ cell^{\mathbb{N}}(\mathsf{inr}\ 5)\ \mathsf{in} \\
&\qquad \mathsf{using}\ y(\\
&\qquad m \leftarrow \mathsf{sget}; \\
&\qquad \mathsf{let}\ m' \ = \ m + m\ \mathsf{in} \\
&\qquad \mathsf{sput}_{m'}; \\
&\qquad \mathsf{return}(m' + m))
\end{aligned}$$

We have explored effects without sessions, but our effects work with sessions seamlessly, as demonstrated by the next example.

Example 10 (Worker-Pool Server)**.** We consider a server providing services over sessions; each session is of type X_S. Moreover, each serving will induce effects Θ (such as database IO) on the server which are shared across sessions. For simplicity, the service is to receive a number, increment it by one, and send it back. Therefore, we define $X_S := ?\mathbb{N}.!\mathbb{N}.\mathsf{end}_?$, and such a session can be served by the following program. It does not use any effect, but we write Θ for generality.

$$\begin{aligned}
&x : X_S \mid \Theta \vdash \mathsf{serve} : \mathbf{1} \\
&\qquad \mathsf{serve} := \mathsf{let}\ (n, x) \ = \ \mathsf{recv}\ x\ \mathsf{in} && \text{(receive number } n\text{)} \\
&\qquad\qquad \mathsf{let}\ x \ = \ \mathsf{send}\ (n+1)\ x\ \mathsf{in} && \text{(send } n+1\text{)} \\
&\qquad\qquad \mathsf{return}(\mathsf{terminate}\ x) && \text{(terminate the session and return)}
\end{aligned}$$

Moreover, we define

$$\Omega := \mathsf{get} : (\mathbf{1} : \mathbf{1} + X_S)$$

which allows fetching new sessions to be served. Similar to Cell^X, the response could be empty due to unavailability of new sessions.

We then define the worker, which repeatedly gets a new session and serve it; note that the loop body always gives inl $\star$, indicating continuing the loop.

$$\begin{aligned}\Gamma \mid \Theta, \Omega \vdash & \mathsf{worker} : \mathbf{1}\\ \mathsf{worker} := & \mathsf{let}\ e \ = \ \star\ \mathsf{in}\\ & \qquad \mathsf{doUntil}\ e.(x \leftarrow \mathsf{sget};\ \mathsf{serve};\ \mathsf{return}(\mathsf{inl}\ \star))\end{aligned}$$

Finally we define workers to spawn two (for simplicity) workers. Note that Γ must be unlimited for the two workers to share; intuitively it contains some global parameters of the server.

$$\begin{aligned}\Gamma \mid \Theta, \Omega \vdash & \mathsf{workers} : \mathbf{1}\\ \mathsf{workers} := & \mathsf{spawn\ worker;\ spawn\ worker}\end{aligned}$$

Note that the two workers race to get new sessions. For each session, it is non-deterministic which worker will serve it.

The above example omits the handling of the get-effects. A natural coemitter to consider is cell^{X_S}; recall that cell is linear and can store sessions. In this scenario, other effectful computations would have put in their effect environments, so they can put sessions to be served to the cell^{X_S}.

Using allows us to add coemitters to an effect environment. This is, however, problematic if those coemitters are instances of the same definition (e.g. multiple instances of cell), because they would have same effect names. By the current definition of Using, Θ (the new coemitter) would overshadow Ω (the existing effect environment). The following example provides an alternative if shadowing is not desired.

Example 11 (Multiple Instances)**.** To avoid shadowing, the simplest solution is to rename the effect names, and the simplest renaming is prefixing. We assume a monoidal structure on effect names where _ is the empty effect name and . is effect name concatenation. Let $S := \Diamond^{\Theta} D$ and define

$$\begin{aligned}\forall \mathsf{i} : (A : B) \in \Theta, z : S, a' : A \vdash & Q_{j.i} := \mathsf{coinst}(\mathsf{emit}^{\mathsf{i}}(a'))(z) : B \otimes S\\ & \vdash \mathsf{prefix}^{j} := \lambda y.\,\mathsf{runner}\{y, za'.Q, f.f\}\\ & :\Diamond^{\Theta} D \multimap \Diamond^{\mathsf{j}.\Theta} \Diamond^{\Theta} D\end{aligned}$$

where $\mathsf{j}.\Theta$ is defined to be same as Θ but with all effect names prefixed with j. The program prefix^{j} takes a Θ-coemitter and gives a $j.\Theta$-coemitter. The latter notably still returns the left-over of Θ-coemitter to be used further, which is vital for the following syntactic sugar:

$$\frac{\Gamma \mid \Omega \vdash P : \Diamond^{\Theta} E \qquad \Delta \mid \mathsf{j}.\Theta, \Omega \vdash Q : D}{\begin{array}{c}\Gamma, \Delta \mid \Omega \vdash \mathsf{using}\ \mathsf{j}.P(Q) :=\\ (d, y) \leftarrow \mathsf{using\ prefix}^{\mathsf{j}}(P)(Q);\ \ \mathsf{return}((d, \mathsf{counit}\ y)) : D \otimes \Diamond^{\Theta} E\end{array}}$$

It is a prefixed version of Using, because Q accesses effects in Θ under the prefix j specified by the user. Note that the returned left-over of the Θ-coemitter is not prefixed, as it is good practice to keep the prefixing local to Q. Also note the possibility of $j.\Theta$ still overshadowing Ω, in which case we take it as the user's intention to locally override.

With example 11 and example 8, we are able to express dining philosophers (see example 19 in section 3.D), where each chopstick is represented by a cell$^{\mathbf{1}}$.

3.8 Related and future Work

Shared States The work of Rocha and Caires [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 is also confluent.

Compared to that work our approach through fixed points is more parsimonious and follows Linear Logic more closely. Their state is a special case (example 18) of our coemitter, and cannot store sessions, which is arguably restrictive in the context of session-based concurrency.

Effects and Linearity Tranquilli [2010] extended simply typed lambda calculus (STLC) with memory access as effects, which is translated to STLC with products, where effectful computations are translated using a state monad; STLC with products is then translated to linear logic proof nets. Orchard and Yoshida [2016] established mutual translations betwee effectful PCF and session-typed π-calculus; note that the latter is not motivated by linear logic. Hasegawa [2002] gave a fully complete CPS translation from computational lambda calculus to linear lambda calculus.

Moggi [1991] introduced a *computational metalanguage* where effectful computations are interpreted as elements of a strong monad. It was later shown [Benton and Wadler, 1996] that every model of intuitionistic linear logic is also a model of effectful computation. However, such models are too special, in that they disregard the ordering of effects. The Enriched Effect Calculus [Egger et al., 2009] addressed this issue by extending Moggi [1991] with linear connectives, in order to express linear paradigms such as linear continuations and linear state. Later Egger et al. [2010], Møgelberg and Staton [2014] showed that every monad in EEC corresponds to a linear state monad.

Linearity of Continuation In our Handler rule, the continuation z' is linear. One can change the functors in section 3.3 to

$$F^{\bullet}_{C}(X) := C \qquad F^{\mathsf{i}:(A:B),\Theta}_{C}(X) := F^{\Theta}_{C}(X) \oplus (A \otimes !(B \parr X))$$

and the exponential ! around $B \parr X$ then allows for discarding and duplication. As a result, z' in Handler would have a normal (non-linear) continuation type $B \to S$ and thus one would be able to express exceptions and non-determinism. On the other hand, the returning value of Q in Runner would be tricky, because $?(B \otimes S)$ is not expressible in CEGV.

In summary, runners and normal handlers are *not* dual: the former is linear while the latter is not. This is also observed by Ahman and Bauer [2020], who argued that runners are more suitable to model resources, including the external world. We also note that a non-linear handler is *not* a generalization of our handler, since the user program in the former would be required to provide non-linear continuations that are discardable and duplicable, which would forbid the use of sessions in the continuation.

Another alternative to linear continuations are affine continuations, which many have argued have benefits over normal continuations, including implementation efficiency [Dolan et al., 2015, 2017] and better reasoning about system resources [de Vilhena and Pottier, 2021]. These benefits apply to linear continuations as well. Note that *op. cit.* uses the word ‘linear’ or ‘one-shot’ to mean ‘affine’. The functor modifications sketched above can be adapted to affine continuations as well.

Multiple Handlers In many effect systems [Pretnar, 2015, Convent et al., 2020, Brachthäuser et al., 2020a], an effectful computation is handled by being wrapped in multiple layers of handlers, each handling a subset of the effects and eliminating them from the effectful computation type. Moreover, inner handlers can emit effects to be handled by outer handlers.

The first feature is related to our merge (example 3). A notable difference is that coemitters after merge are still independent (theorem 19), but layers of handlers in existing effect systems might interfere with each other in the presense of normal continuations (e.g., exceptions and non-determinisms). Different orderings of handlers give different semantics, sometimes none of which is the desired one. To solve this issue, scoped effects [Wu et al., 2014, Piróg et al., 2018, Yang et al., 2022] decouple scopes from handlers. We note that it is unclear how to derive a version of merge for normal continuations, but might provide new perspectives on scoped effects.

The second feature is related to our effects translation (example 7). Finally, we note that both features are primitive in those effect systems, while derived in our system.

Handler Instances Bauer and Pretnar [2015] introduces instances associated with resources which are essentially runners. However, their handler type does not describe effects, and thus is not intended for effect safety. Leijen [2018] introduces references which can be created at runtime and referred to using variable names, but breaks effect-safety because the variable names of cells might escape the handler. XIE et al. [2022] solves the escaping issue by using rank-2 polymorphism to encapsulate cell variables. Biernacki et al. [2019] assign names to the handle binder so that effects in scope can specify the handle binder they intend; they still require special treatments to prevent effects from escaping their scope. While the mentioned works focus on preventing effects from escaping handlers and breaking effect safety, our Using (and the prefixed version in example 11) rule allows for safe escaping of the coemitter.

Appendices

3.A πLL, full specification

In this appendix we report the full specification of $\pi\mathsf{LL}$ with minor changes to adapt it to the notation used in this paper. We refer the interested reader to Montesi and Peressotti [2021] for more details.

Syntax and typing

$P, Q := x[y].\, P$	*output y on x and continue as P*
$\mid x(y).\, P$	*input y on x and continue as P*
$\mid x[].\, P$	*output (empty message) on x and continue as P*
$\mid x().\, P$	*input (empty message) on x and continue as P*
$\mid x[\textsc{l}].\, P$	*select left on x and continue as P*
$\mid x[\textsc{r}].\, P$	*select right on x and continue as P*
$\mid y.\mathsf{case}\{\textsc{l}{:}P, \textsc{r}{:}Q\}$	*offer on x a choice to continue as P (left) and Q (right)*
$\mid x[\textsc{disp}].\, P$	*request to dispose a replicable process on x and continue as P*
$\mid x[\textsc{dup}](x').\, P$	*request to duplicate a replicable process on x and bind the copy to x' in P*
$\mid x[\textsc{use}].\, P$	*consume a replicable process and continue as P*
$\mid !x.\{P\}$	*provider of replicas of process P*
$\mid x[X].\, P$	*output type A on x and continue as P*
$\mid x(A).\, P$	*input a type on x as X and continue as P*
$\mid \mathbf{0}$	*terminated process*
$\mid P \mid Q$	*parallel composition of P and Q*
$\mid \nu xy\, P$	*session with endpoints x and y in P*
$\mid x \leftrightarrow y$	*forwarding of x and y*

$A, B := A \otimes B$	*send A, continue as B*
$\mid A \parr B$	*receive A, continue as B*
$\mid \mathbf{1}$	*send close, unit for $\otimes$*
$\mid \bot$	*receive close, unit for $\parr$*
$\mid A \oplus B$	*select A or B*
$\mid A \mathbin{\&} B$	*offer A or B*
$\mid ?A$	*request a replicable A*
$\mid !A$	*provide a replicable A*
$\mid \exists X.A$	*send a type and use it for X in A*
$\mid \forall X.A$	*receive a type and use it for X in A*
$\mid X$	*type variable*
$\mid X^\perp$	*dual of a type variable*

$$(A \otimes B)^\perp = A^\perp \parr B^\perp \qquad (A \parr B)^\perp = A^\perp \parr B^\perp \qquad \mathbf{1}^\perp = \bot \qquad \bot^\perp = \mathbf{1}$$

$$(A \oplus B)^\perp = A^\perp \mathbin{\&} B^\perp \qquad (A \mathbin{\&} B)^\perp = A^\perp \oplus \& B \qquad (?A)^\perp = !A^\perp$$

$$(!A)^\perp = ?A^\perp$$

$$(\exists X.A)^\perp = \forall X.A^\perp \qquad (\forall X.A)^\perp = \exists X.A^\perp \qquad (X)^\perp = X^\perp \qquad (X^\perp)^\perp = X$$

$$\textsc{HMix0}\ \dfrac{}{\mathbf{0} \vdash \emptyset} \qquad \textsc{HMix}\ \dfrac{P \vdash \mathcal{G} \qquad Q \vdash \mathcal{H}}{P \mid Q \vdash \mathcal{G} \mid \mathcal{H}} \qquad \textsc{Cut}\ \dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : A^\perp}{\nu xy\, P \vdash \mathcal{G} \mid \Gamma, \Delta}$$

$$\textsc{Ax}\ \dfrac{}{x \leftrightarrow y \vdash x : A^\perp, y : A} \qquad \textsc{Tensor}\ \dfrac{P \vdash \Gamma, y : A \mid \Delta, x : B}{x[y].\, P \vdash \Gamma, \Delta, x : A \otimes B} \qquad \textsc{One}\ \dfrac{P \vdash \emptyset}{x[].\, P \vdash x : \mathbf{1}}$$

$$\textsc{Par}\ \dfrac{P \vdash \Gamma, y : A, x : B}{x(y).\, P \vdash \Gamma, x : A \parr B} \qquad \textsc{Bot}\ \dfrac{P \vdash \Gamma}{x().\, P \vdash \Gamma, x : \bot} \qquad \textsc{PlusL}\ \dfrac{P \vdash \Gamma, x : A}{x[\textsc{l}].\, P \vdash \Gamma, x : A \oplus B}$$

$$\textsc{PlusR}\ \dfrac{P \vdash \Gamma, x : B}{x[\textsc{r}].\, P \vdash \Gamma, x : A \oplus B} \qquad \textsc{With}\ \dfrac{P \vdash \Gamma, x : A \qquad Q \vdash \Gamma, x : B}{x.\mathsf{case}\{\textsc{l}{:}P, \textsc{r}{:}Q\} \vdash \Gamma, x : A \mathbin{\&} B}$$

$$\textsc{WhyW}\ \dfrac{P \vdash \Gamma}{x[\textsc{disp}].\, P \vdash \Gamma, x : ?A} \qquad \textsc{WhyD}\ \dfrac{P \vdash \Gamma, x : A}{x[\textsc{use}].\, P \vdash \Gamma, x : ?A} \qquad \textsc{WhyC}\ \dfrac{P \vdash \Gamma, x : ?A, x' : ?A}{x[\textsc{dup}](x').\, P \vdash \Gamma, x : ?A}$$

$$\textsc{OfC}\ \dfrac{P \vdash ?\Gamma, x : A}{!x.\{P\} \vdash ?\Gamma, x : !A} \qquad \textsc{Exists}\ \dfrac{P \vdash \Gamma, x : A\{B/X\}}{x[B].\, P \vdash \Gamma, x : \exists X.A} \qquad \textsc{Forall}\ \dfrac{P \vdash \Gamma, x : A \qquad X \notin \Gamma}{x(X).\, P \vdash \Gamma, x : \forall X.A}$$

SOS Specification for typing derivations

$$\frac{P \vdash \emptyset}{x[].\,P \vdash x : \mathbf{1}} \xrightarrow{x[]} P \vdash \emptyset \qquad \frac{P \vdash \Gamma}{x().\,P \vdash \Gamma, x : \bot} \xrightarrow{x()} P \vdash \Gamma$$

$$\frac{P \vdash \Gamma, x : A \mid \Delta, y : B}{x[y].\,P \vdash \Gamma, \Delta, x : B \otimes A} \xrightarrow{x[y]} P \vdash \Gamma, x : A \mid \Delta, y : B$$

$$\frac{P \vdash \Gamma, x : A, y : B}{x(y).\,P \vdash \Gamma, x : B \parr A} \xrightarrow{x(y)} P \vdash \Gamma, x : A, y : B$$

$$\frac{P \vdash \Gamma, x : A}{x[\textsc{l}].\,P \vdash \Gamma, x : A \oplus B} \xrightarrow{x[\textsc{l}]} P \vdash \Gamma, x : A$$

$$\frac{P \vdash \Gamma, x : B}{x[\textsc{r}].\,P \vdash \Gamma, x : A \oplus B} \xrightarrow{x[\textsc{r}]} P \vdash \Gamma, x : B$$

$$\frac{P \vdash \Gamma, x : A \qquad Q \vdash \Gamma, x : B}{x.\mathsf{case}\{\textsc{l}{:}P, \textsc{r}{:}Q\} \vdash \Gamma, x : A \,\&\, B} \xrightarrow{x(\textsc{l})} P \vdash \Gamma, x : A$$

$$\frac{P \vdash \Gamma, x : A \qquad Q \vdash \Gamma, x : B}{x.\mathsf{case}\{\textsc{l}{:}P, \textsc{r}{:}Q\} \vdash \Gamma, x : A \,\&\, B} \xrightarrow{x(\textsc{r})} Q \vdash \Gamma, x : B$$

$$\frac{P \vdash \Gamma}{x[\textsc{disp}].\,P \vdash \Gamma, x : ?A} \xrightarrow{x[\textsc{disp}]} \frac{P \vdash \Gamma}{x().\,P \vdash \Gamma, x : \bot}$$

$$\frac{P \vdash \Gamma, x : A}{x[\textsc{use}].\,P \vdash \Gamma, x : ?A} \xrightarrow{x[\textsc{use}]} P \vdash \Gamma, x : A$$

$$\frac{P \vdash \Gamma, x : ?A, x' : ?A}{x[\textsc{dup}](x').\,P \vdash \Gamma, x : ?A} \xrightarrow{x[\textsc{dup}]} \frac{P \vdash \Gamma, x : ?A, x' : ?A}{x(x').\,P \vdash \Gamma, x : ?A \parr ?A}$$

$$\frac{P \vdash \Gamma, x : A}{!x.\{P\} \vdash \Gamma, x : !A} \xrightarrow{x(\textsc{use})} P \vdash \Gamma, x : A$$

$$\frac{\textsc{Fn}(P) \setminus \{x\} = \{z_1, \ldots, z_n\}}{\dfrac{P \vdash ?\Gamma, x : A}{!x.\{P\} \vdash ?\Gamma, x : !A} \xrightarrow{x(\textsc{disp})} \dfrac{\dfrac{\mathbf{0} \vdash \emptyset}{x[].\,\mathbf{0} \vdash x : \mathbf{1}}}{z_1[\textsc{disp}].\ldots.z_n[\textsc{disp}].\,x[].\,\mathbf{0} \vdash ?\Gamma, x : \mathbf{1}}}$$

$$\dfrac{\textsc{Fn}(P) \setminus \{x\} = \{z_1, \dots, z_n\} \qquad \textsc{Fn}(P\sigma) \cap \textsc{Fn}(P) = \emptyset}{\dfrac{P \vdash ?\Gamma, x : A}{!x.\{P\} \vdash ?\Gamma, x : !A} \xrightarrow{x(\text{DUP})} \dfrac{\dfrac{\dfrac{\dfrac{P \vdash ?\Gamma, x : A}{!x.\{P\} \vdash ?\Gamma, x : !A} \qquad \dfrac{P\sigma \vdash ?\Gamma\sigma, x\sigma : A}{!x.\{P\}\sigma \vdash ?\Gamma\sigma, x\sigma : !A}}{!x.\{P\} \mid !x.\{P\}\sigma \vdash ?\Gamma, x : !A \mid ?\Gamma\sigma, x\sigma : !A}}{x[x\sigma].\,(!x.\{P\} \mid !x.\{P\}\sigma) \vdash ?\Gamma, ?\Gamma\sigma, x : !A \otimes !A}}{z_1[\text{DUP}](z_1\sigma)\dots z_n[\text{DUP}](z_n\sigma).\,x[x\sigma].\,(!x.\{P\} \mid !x.\{P\}\sigma) \vdash ?\Gamma, x : !A \otimes !A}}$$

Par0
$$\dfrac{P \vdash \mathcal{G} \xrightarrow{l} P' \vdash \mathcal{G}' \qquad \textsc{Bn}(l) \cap \textsc{Fn}(Q) = \emptyset}{\dfrac{P \vdash \mathcal{G} \qquad Q \vdash \mathcal{H}}{P \mid Q \vdash \mathcal{G} \mid \mathcal{H}} \xrightarrow{l} \dfrac{P' \vdash \mathcal{G}' \qquad Q \vdash \mathcal{H}}{P' \mid Q \vdash \mathcal{G}' \mid \mathcal{H}}}$$

Par1
$$\dfrac{Q \vdash \mathcal{H} \xrightarrow{l} Q' \vdash \mathcal{H}' \qquad \textsc{Bn}(l) \cap \textsc{Fn}(P) = \emptyset}{\dfrac{P \vdash \mathcal{G} \qquad Q \vdash \mathcal{H}}{P \mid Q \vdash \mathcal{G} \mid \mathcal{H}} \xrightarrow{l} \dfrac{P \vdash \mathcal{G} \qquad Q \vdash \mathcal{H}}{P \mid Q' \vdash \mathcal{G} \mid \mathcal{H}'}}$$

Syn
$$\dfrac{P \vdash \mathcal{G} \xrightarrow{l} P' \vdash \mathcal{G}' \qquad Q \vdash \mathcal{H} \xrightarrow{l} Q' \vdash \mathcal{H}' \qquad \textsc{Bn}(l \mid l') \cap \textsc{Fn}(P \mid Q) = \emptyset}{\dfrac{P \vdash \mathcal{G} \qquad Q \vdash \mathcal{H}}{P \mid Q \vdash \mathcal{G} \mid \mathcal{H}} \xrightarrow{l} \dfrac{P \vdash \mathcal{G} \qquad Q \vdash \mathcal{H}}{P \mid Q' \vdash \mathcal{G} \mid \mathcal{H}'}}$$

AEQ
$$\dfrac{P =_\alpha Q \qquad Q \vdash \mathcal{G} \xrightarrow{l} Q' \vdash \mathcal{G}'}{P \vdash \mathcal{G} \xrightarrow{l} Q' \vdash \mathcal{G}'}$$

Res
$$\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : A^\perp \xrightarrow{l} P' \vdash \mathcal{G}' \mid \Gamma', x : A \mid \Delta', y : A^\perp \qquad x, y \notin \mathrm{N}(l)}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : A^\perp}{\nu xy\, P \vdash \mathcal{G} \mid \Gamma, \Delta} \xrightarrow{l} \dfrac{P' \vdash \mathcal{G}' \mid \Gamma', x : A \mid \Delta', y : A^\perp}{\nu xy\, P' \vdash \mathcal{G}' \mid \Gamma', \Delta'}}$$

Link
$$\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A \mid y : A^\perp, z : A \xrightarrow{y \leftrightarrow z} P' \vdash \mathcal{G} \mid \Gamma, x : A}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A \mid y : A^\perp, z : A}{\nu xy\, P \vdash \mathcal{G} \mid \Gamma} \xrightarrow{\tau} P'\{x/z\} \vdash \mathcal{G} \mid \Gamma, x : A}$$

One-Bot
$$\dfrac{P \vdash \mathcal{G} \mid x : \mathbf{1} \mid \Gamma, y : \bot \xrightarrow{x[] \mid y()} P' \vdash \mathcal{G} \mid \Gamma}{\dfrac{P \vdash \mathcal{G} \mid x : \mathbf{1} \mid \Gamma, y : \bot}{\nu xy\, P \vdash \mathcal{G} \mid \Gamma} \xrightarrow{\tau} P' \vdash \mathcal{G} \mid \Gamma}$$

Tensor-Par

$$\frac{P \vdash \mathcal{G} \mid \Gamma, \Delta x : A \otimes B \mid \Sigma, y : A^\perp \parr B^\perp \xrightarrow{x[x'] | y(y')} P' \vdash \mathcal{G} \mid \Gamma, x' : A \mid \Delta, x : B \mid \Sigma, y' : A^\perp, y : B^\perp}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, \Delta x : A \otimes B \mid \Sigma, y : A^\perp \parr B^\perp}{\nu xy\, P \vdash \mathcal{G} \mid \Gamma, \Delta, \Sigma} \xrightarrow{\tau} \dfrac{\dfrac{P \vdash \mathcal{G} \mid \Gamma x' : A \mid \Delta, x : A \mid \Sigma, y' : A^\perp, y : B^\perp}{\nu x'y'\, P \vdash \mathcal{G} \mid \Gamma, \Delta x : B \mid \Sigma, y : B^\perp}}{\nu xy\, \nu x'y'\, P' \vdash \mathcal{G} \mid \Gamma, \Delta, \Sigma}}$$

PlusL-With

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x : A \oplus B \mid \Delta, y : A^\perp \mathbin{\&} B^\perp \xrightarrow{x[\text{L}] | y(\text{L})} P' \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : A^\perp}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A \oplus B \mid \Delta, y : A^\perp \mathbin{\&} B^\perp}{\nu xy\, P \vdash \mathcal{G} \mid \Gamma, \Delta} \xrightarrow{\tau} \dfrac{P' \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : A^\perp}{\nu xy\, P' \vdash \mathcal{G} \mid \Gamma, \Delta}}$$

PlusR-With

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x : A \oplus B \mid \Delta, y : A^\perp \mathbin{\&} B^\perp \xrightarrow{x[\text{R}] | y(\text{R})} P' \vdash \mathcal{G} \mid \Gamma, x : B \mid \Delta, y : B^\perp}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A \oplus B \mid \Delta, y : A^\perp \mathbin{\&} B^\perp}{\nu xy\, P \vdash \mathcal{G} \mid \Gamma, \Delta} \xrightarrow{\tau} \dfrac{P' \vdash \mathcal{G} \mid \Gamma, x : B \mid \Delta, y : B^\perp}{\nu xy\, P' \vdash \mathcal{G} \mid \Gamma, \Delta}}$$

WhyD-OfC

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x : ?A \mid \Delta, y : !A^\perp \xrightarrow{x[\text{USE}] | y(\text{USE})} P' \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : A^\perp}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : ?A \mid \Delta, y : !A^\perp}{\nu xy\, P \vdash \mathcal{G} \mid \Gamma, \Delta} \xrightarrow{\tau} \dfrac{P' \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : A^\perp}{\nu xy\, P' \vdash \mathcal{G} \mid \Gamma, \Delta}}$$

WhyC-OfC

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x : ?A \mid \Delta, y : !A^\perp \xrightarrow{x[\text{DUP}]().\, | y(\text{DUP})} P' \vdash \mathcal{G} \mid \Gamma, x : ?A \parr ?A \mid \Delta, y : A^\perp \otimes A^\perp}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : ?A \mid \Delta, y : !A^\perp}{\nu xy\, P \vdash \mathcal{G} \mid \Gamma, \Delta} \xrightarrow{\tau} \dfrac{P' \vdash \mathcal{G} \mid \Gamma, x : ?A \parr ?A \mid \Delta, y : A^\perp \otimes A^\perp}{\nu xy\, P' \vdash \mathcal{G} \mid \Gamma, \Delta}}$$

WhyW-OfC

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x : ?A \mid \Delta, y : !A^\perp \xrightarrow{x[\text{DISP}] | y(\text{DISP})} P' \vdash \mathcal{G} \mid \Gamma, x : \bot \mid \Delta, y : \mathbf{1}}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : ?A \mid \Delta, y : !A^\perp}{\nu xy\, P \vdash \mathcal{G} \mid \Gamma, \Delta} \xrightarrow{\tau} \dfrac{P' \vdash \mathcal{G} \mid \Gamma, x : \bot \mid \Delta, y : \mathbf{1}}{\nu xy\, P' \vdash \mathcal{G} \mid \Gamma, \Delta}}$$

Exists-Forall

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x : \exists X.A \mid \Delta, y : \forall X.A^\perp \xrightarrow{x[B] | y(B)} P' \vdash \mathcal{G} \mid \Gamma, x : A\{B/X\} \mid \Delta, y : A\{B/X\}^\perp}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : \exists X.A \mid \Delta, y : \forall X.A^\perp}{\nu xy\, P \vdash \mathcal{G} \mid \Gamma, \Delta} \xrightarrow{\tau} \dfrac{P' \vdash \mathcal{G} \mid \Gamma, x : A\{B/X\} \mid \Delta, y : A\{B/X\}^\perp}{\nu xy\, P' \vdash \mathcal{G} \mid \Gamma, \Delta}}$$

3.B Omitted content of CELL

Omitted environment transition rules of CELL are in fig. 3.B.1,fig. 3.B.2, fig. 3.B.3.

$$\Gamma, x : \boxdot^{\Theta} C \xrightarrow[\textsc{BoWW}]{\boxdot x[]} \Gamma, x : C$$

$$\Delta, \Gamma, x : \boxdot^{\Theta, \mathsf{i}:(A:B)} C \xrightarrow[\textsc{BoAA}]{\boxdot x^{\mathsf{i}}[a]} \Delta, a : A \mid \Gamma, x : B \parr \boxdot^{\Theta, \mathsf{i}:(A:B)} C$$

$$\Gamma, y : \Diamonddot^{\Theta} C \xrightarrow[\textsc{DiW}]{\Diamonddot y()} \Gamma, y : C \qquad\qquad \Gamma, y : \Diamonddot^{\Theta} C \xrightarrow[\textsc{DiA}]{\Diamonddot y^{\mathsf{i}}(a)} \Gamma, a' : A, y : B \otimes \Diamonddot^{\Theta} C$$

$$\textsc{BoW-DiW}\quad \dfrac{\mathcal{G} \mid \Gamma, x : \boxdot^{\Theta} C \mid \Delta, y : \Diamonddot^{\Theta^{\perp}} C^{\perp} \xrightarrow{\boxdot x[] \mid \Diamonddot y()} \mathcal{G} \mid \Gamma, x : C \mid \Delta, y : C^{\perp}}{\mathcal{G} \mid \Gamma, \Delta \xrightarrow{\tau} \mathcal{G} \mid \Gamma, \Delta}$$

$$\textsc{BoA-DiA}\quad \dfrac{\begin{array}{c}\mathcal{G} \mid \Gamma_0, \Gamma_1, x : \boxdot^{\Theta} C \mid \Delta, y : \Diamonddot^{\Theta^{\perp}} C^{\perp} \xrightarrow{\boxdot x^{\mathsf{i}}[a] \mid \Diamonddot y^{\mathsf{i}}(a')} \\ \mathcal{G} \mid \Gamma_0, a : A \mid \Gamma_1, x : B \parr \boxdot^{\Theta} C \mid \Delta, a' : A^{\perp}, y : B^{\perp} \otimes \Diamonddot^{\Theta^{\perp}} C^{\perp} \qquad \mathsf{i} : (A : B) \in \Theta\end{array}}{\mathcal{G} \mid \Gamma_0, \Gamma_1, \Delta \xrightarrow{\tau} \mathcal{G} \mid \Gamma_0, \Gamma_1, \Delta}$$

Figure 3.B.1: Effects, transition rules for environments

The following process can be thought of as an internalized BoA: it 'absorbs' a request/response interaction into the effect type $\boxdot^{\Theta} C$.

Proposition 29 (Absorb)**.** *It is provable that* $(A \otimes B \parr \boxdot^{\Theta} C) \multimap \boxdot^{\Theta} C$. *More concretely, we have* $\mathsf{absorb}_{a'yx} \vdash a' : A^{\perp}, y : B^{\perp} \otimes \Diamonddot^{\Theta^{\perp}} C^{\perp}, x : \boxdot^{\Theta} C$.

Proof.

$$\dfrac{a' \leftrightarrow a \mid x \leftrightarrow y \vdash a' : A^{\perp}, a : A \mid y : B^{\perp} \otimes \Diamonddot^{\Theta^{\perp}} C^{\perp}, x : B \parr \boxdot^{\Theta} C}{\mathsf{absorb}_{a'yx} := \boxdot x^{\mathsf{i}}[a].\, a' \leftrightarrow a \mid x \leftrightarrow y \vdash a' : A^{\perp}, y : B^{\perp} \otimes \Diamonddot^{\Theta^{\perp}} C^{\perp}, x : \boxdot^{\Theta} C}\ \textsc{BoA}$$

□

Proposition 30 (Relaxing)**.** *It is derivable that* $\boxdot^{\Theta} C \multimap \boxdot^{\Theta, i:(A:B)} C$. *More concretely, there is a process* $\mathsf{relax}_{yx} \vdash y : \Diamonddot^{\Theta^{\perp}} C^{\perp}, x : \boxdot^{\Theta'} C$ *where* $\Theta' := \Theta, i : (A : B)$.

BoCP

$$\frac{x_0, x_1 \notin \textsc{Fn}(l) \qquad \mathcal{G} \mid \Gamma, x_0 : \boxdot^{\Theta} C_0 \mid \Delta, x_1 : \boxdot^{\Theta} C_1 \xrightarrow{l} \mathcal{G}' \mid \Gamma', x_0 : \boxdot^{\Theta} C_0 \mid \Delta', x_1 : \boxdot^{\Theta} C_1}{\mathcal{G} \mid \Gamma, \Delta, x : \boxdot^{\Theta}(C_0 \otimes C_1) \xrightarrow{l} \mathcal{G}' \mid \Gamma', \Delta', x : \boxdot^{\Theta}(C_0 \otimes C_1)}$$

BoCA0

$$\frac{x_0, x_1 \notin \textsc{Fn}(l) \qquad \mathsf{i} : (A : B) \in \Theta \qquad \mathcal{G} \mid \Gamma_0, \Gamma_1, x_0 : \boxdot^{\Theta} C_0 \mid \Delta, x_1 : \boxdot^{\Theta} C_1 \xrightarrow{l \mid \boxdot x_0{}^{\mathsf{i}}[a]} \mathcal{G}' \mid \Gamma_0, a : A \mid \Gamma_1, x_0 : B \parr \boxdot^{\Theta} C_0 \mid \Delta', x_1 : \boxdot^{\Theta} C_1}{\mathcal{G} \mid \Gamma_0, \Gamma_1, \Delta, x : \boxdot^{\Theta}(C_0 \otimes C_1) \xrightarrow{l \mid \boxdot x^{\mathsf{i}}[a]} \mathcal{G} \mid \Gamma_0, a : A \mid \Gamma_1, \Delta', x : B \parr \boxdot^{\Theta}(C_0 \otimes C_1)}$$

BoCW

$$\frac{x_0, x_1 \notin \textsc{Fn}(l) \qquad \mathcal{G} \mid \Gamma, x_0 : \boxdot^{\Theta} C_0 \mid \Delta, x_1 : \boxdot^{\Theta} C_1 \xrightarrow{l \mid \boxdot x_0[] \mid \boxdot x_1[]} \mathcal{G}' \mid \Gamma, x_0 : C_0 \mid \Delta, x_1 : C_1}{\mathcal{G} \mid \Gamma, \Delta, x : \boxdot^{\Theta}(C_0 \otimes C_1) \xrightarrow{l \mid \boxdot x[]} \mathcal{G}' \mid \Gamma, \Delta, x : C_0 \otimes C_1}$$

Figure 3.B.2: Races, transition rules for environments

$$\Gamma, e : E^{\perp}, x : \boxdot^{\Theta} C \xrightarrow[\textsc{BoRR}]{\tau} \Gamma, e : E^{\perp}, x : \boxdot^{\Theta} C$$

Figure 3.B.3: Do-Until, transition rules for environments

Proof. Let $S := \Diamond^{\Theta'^{\perp}} C^{\perp}$.

$$\dfrac{\dfrac{}{P := x \leftrightarrow i \vdash x : S^{\perp}, i : S}\ \textsc{Ax} \qquad \mathsf{i} : (A : B) \in \Theta \subset \Theta'}{Q := \mathsf{absorb}_{a'z'z} \vdash z : S^{\perp}, a' : A^{\perp}, z' : B^{\perp} \otimes S}\ \textsc{proposition 29}$$

$$\dfrac{Q \qquad \dfrac{\dfrac{}{f \leftrightarrow y \vdash f : C, y : C^{\perp}}\ \textsc{Ax}}{R := \boxdot f[].\, f \leftrightarrow y \vdash f : S^{\perp}, y : C^{\perp}}\ \textsc{BoW}}{\mathsf{relax}_{yx} := \Diamond y\{i.\, P, za'z'.\, Q, f.\, R\} \vdash y : \Diamond^{\Theta^{\perp}} C^{\perp}, x : \boxdot^{\Theta'} C}\ \textsc{Di}$$

□

Example 12 (Nullary handler)**.** For the sake of completeness, we give the nullary case of merge.

$$\dfrac{P \vdash \Gamma, i : \mathbf{1} \qquad f \leftrightarrow y \vdash f : \mathbf{1}^{\perp}, y : \mathbf{1}}{\mathsf{merge0}_y(i.P) := \Diamond y\{i.\, P, za'z''.\, \_, f.\, f \leftrightarrow y\} \vdash \Gamma, y : \Diamond \bullet \mathbf{1}}\ \textsc{Di}$$

Proof of theorem 19. By rule DiA we have that

$$\mathsf{merge}_y(y_1.y_1[y_0].\, P) = \Diamond y\{y_1.\, y_1[y_0].\, P, za'z''.\, \vec{Q}, f.\, f \leftrightarrow y\} \xrightarrow{\Diamond y^{\mathsf{i}}(a')} Z$$

where

$$Z = \nu i'z''\, \nu y_1 z\, (z(z').\, \boxdot z'^{\mathsf{i}}[a].\, (a \leftrightarrow a' \mid z'(b).\, z''\langle b\rangle.\, z''\langle z'\rangle.\, z \leftrightarrow z'') \mid y_1[y_0].\, P \mid$$
$$\mid i'(b').\, y\langle b'\rangle.\, \Diamond y\{i.\, i \leftrightarrow i', za'z''.\, \vec{Q}, f.\, f \leftrightarrow y\}).$$

Let X be $\nu y_1 y_1'\, \nu y_0 y_0'\, (P' \mid y_0'(b').\, y\langle b'\rangle.\, \mathsf{merge}_y(i.i\langle y_0'\rangle.\, i \leftrightarrow y_1'))$. We need to prove that $Z \approx X$. The interleaved execution of the various parallel components of X and Z leads to an explosion of cases. To keep the proof manageable, we establish a correspondence between traces that subsume the remaining ones by theorem 34, theorem 26, lemma 27. For each transition, we report the most significant rules omitting rules AEQ, Par0, Par1, and Res where their use is clear from the context. Consider the following trace for Z.

$$
\begin{aligned}
Z = {} & \nu i' z''\, \nu y_1 z\, (z(z').\, \boxbox z'^{\mathsf{i}}[a].\, (a \leftrightarrow a' \mid z'(b).\, z''\langle b\rangle.\, z''\langle z'\rangle.\, z \leftrightarrow z'') \mid y_1[y_0].\, P \mid \\
& \quad \mid i'(b').\, y\langle b'\rangle.\, \Diamonddot y\{i.\, i \leftrightarrow i',\, za'z''.\, \vec{Q},\, f.\, f \leftrightarrow y\}) \\
& \downarrow\tau \qquad\qquad \text{By } \textsc{Tensor-Par} \text{ for } y_1, z \\
& \nu i' z''\, \nu y_1 z\, \nu y_0 z'\, (\boxbox z'^{\mathsf{i}}[a].\, (a \leftrightarrow a' \mid z'(b).\, z''\langle b\rangle.\, z''\langle z'\rangle.\, z \leftrightarrow z'') \mid P \mid \\
& \quad \mid i'(b').\, y\langle b'\rangle.\, \Diamonddot y\{i.\, i \leftrightarrow i',\, za'z''.\, \vec{Q},\, f.\, f \leftrightarrow y\}) \\
& \downarrow\tau \qquad\qquad \text{By } \textsc{BoA-Dia} \text{ for } y_0, z' \text{ and } P \xrightarrow{\Diamonddot y_0{}^{\mathsf{i}}(a'')} P' \\
& \nu i' z''\, \nu y_1 z\, \nu a a''\, \nu y_0 z'\, (a \leftrightarrow a' \mid z'(b).\, z''\langle b\rangle.\, z''\langle z'\rangle.\, z \leftrightarrow z'' \mid P' \mid \\
& \quad \mid i'(b').\, y\langle b'\rangle.\, \Diamonddot y\{i.\, i \leftrightarrow i',\, za'z''.\, \vec{Q},\, f.\, f \leftrightarrow y\}) \\
& \downarrow\tau \qquad\qquad \text{By } \textsc{Link} \text{ for } a, a'' \\
Z_1 = {} & \nu i' z''\, \nu y_1 z\, \nu y_0 z'\, (z'(b).\, z''\langle b\rangle.\, z''\langle z'\rangle.\, z \leftrightarrow z'' \mid P' \mid i'(b').\, y\langle b'\rangle.\, \Diamonddot y\{i.\, i \leftrightarrow i',\, za'z''.\, \vec{Q},\, f.\, f \leftrightarrow y\})
\end{aligned}
$$

Next, we need to synchronise over y_0 which might depend on other endpoints in P' before it is ready.

$$
\begin{aligned}
& Z_1 \\
& \downarrow l_1 \qquad\qquad \text{By rule } \textsc{Res} \text{ if } P' \xrightarrow{l_1} P'_1 \text{ and } y_0, y_1 \notin \mathrm{N}(l_1) \\
& \nu i' z''\, \nu y_1 z\, \nu y_0 z'\, (z'(b).\, z''\langle b\rangle.\, z''\langle z'\rangle.\, z \leftrightarrow z'' \mid P'_1 \mid i'(b').\, y\langle b'\rangle.\, \Diamonddot y\{i.\, i \leftrightarrow i',\, za'z''.\, \vec{Q},\, f.\, f \leftrightarrow y\}) \\
& \vdots \\
& \downarrow l_n \qquad\qquad \text{By rule } \textsc{Res} \text{ if } P'_{n-1} \xrightarrow{l_n} P'_n \text{ and } y_0, y_1 \notin \mathrm{N}(l_n) \\
Z_2 = {} & \nu i' z''\, \nu y_1 z\, \nu y_0 z'\, (z'(b).\, z''\langle b\rangle.\, z''\langle z'\rangle.\, z \leftrightarrow z'' \mid P'_n \mid i'(b').\, y\langle b'\rangle.\, \Diamonddot y\{i.\, i \leftrightarrow i',\, za'z''.\, \vec{Q},\, f.\, f \leftrightarrow y\}) \\
& \downarrow\tau \qquad\qquad \text{By } \textsc{Tensor-Par} \text{ for } y_0, z',\ P'_n \xrightarrow{y_0[b'']} P'' \\
& \nu i' z''\, \nu y_1 z\, \nu y_0 z'\, \nu b b''\, (z''\langle b\rangle.\, z''\langle z'\rangle.\, z \leftrightarrow z'' \mid P'' \mid i'(b').\, y\langle b'\rangle.\, \Diamonddot y\{i.\, i \leftrightarrow i',\, za'z''.\, \vec{Q},\, f.\, f \leftrightarrow y\}) \\
& \downarrow\tau \qquad\qquad \text{By } \textsc{Tensor-Par} \text{ for } i', z'' \\
& \downarrow\tau \qquad\qquad \text{By } \textsc{Link} \text{ for } i', z'' \\
& \nu i' z''\, \nu b' b\, \nu y_1 z\, \nu y_0 z'\, (z''\langle z'\rangle.\, z \leftrightarrow z'' \mid P''\{b/b''\} \mid y\langle b'\rangle.\, \Diamonddot y\{i.\, i \leftrightarrow i',\, za'z''.\, \vec{Q},\, f.\, f \leftrightarrow y\}) \\
& \Big\downarrow y[b''] \\
& \downarrow\tau \qquad\qquad \text{By } \textsc{Link} \text{ for } b', b'' \\
Z_3 = {} & \nu i' z''\, \nu y_1 z\, \nu y_0 z'\, (z''\langle z'\rangle.\, z \leftrightarrow z'' \mid P'' \mid \Diamonddot y\{i.\, i \leftrightarrow i',\, za'z''.\, \vec{Q},\, f.\, f \leftrightarrow y\})
\end{aligned}
$$

Next, we need to perform an action over y, any action, we select a general use of an effect j.

$$
\begin{aligned}
& Z_3 \\
& \big\downarrow \diamond y^{\mathsf{j}}(a') \\
& \nu i'z''\, \nu y_1 z\, \nu y_0 z'\, (z''\langle z'\rangle .\, z \leftrightarrow z'' \mid P'' \mid \nu i''z''\, \nu iz\, (Q^{\mathsf{j}} \mid i \leftrightarrow i' \mid \\
& i''(b').\, y\langle b\rangle .\, \diamond y\{i.\, i \leftrightarrow i'', za'z''.\, \vec{Q}, f.\, f \leftrightarrow y\}) \\
& \downarrow\tau \qquad\qquad \text{By } \textsc{Link} \text{ for } i, i' \\
Z_4 = {} & \nu i'z''\, \nu y_1 z\, \nu y_0 z'\, (z''\langle z'\rangle .\, z \leftrightarrow z'' \mid P'' \mid \nu i''z''\, (Q^{\mathsf{j}}\{i'/z\} \mid i''(b').\, y\langle b\rangle .\, \diamond y\{i.\, i \leftrightarrow i'', za'z''.\, \vec{Q}, f.\, f \leftrightarrow y\} \\
= {} & \nu i'z''\, \nu y_1 z\, \nu y_0 z'\, (z''\langle z'\rangle .\, z \leftrightarrow z'' \mid P'' \mid \nu i''z''\, (Q^{\mathsf{j}}\{i'/z\} \mid i''(b').\, y\langle b\rangle .\, \mathsf{merge}_y(i.i \leftrightarrow i''))
\end{aligned}
$$

Then we build a matching trace for X to match the saturation of Z.

$$X = \nu y_1 y_1'\, \nu y_0 y_0'\, (P' \mid y_0'(b').\, y\langle b'\rangle .\, \mathsf{merge}_y(i.i\langle y_0'\rangle .\, i \leftrightarrow y_1'))$$

Similarly to Z_1, we need to synchronise over y_0 which might depend on other endpoints in P' before it is ready.

$$
\begin{aligned}
& X \\
& \downarrow l_1 \qquad\qquad \text{By rule } \textsc{Res} \text{ if } P' \xrightarrow{l_1} P_1' \text{ and } y_0, y_1 \notin \mathrm{N}(l_1) \\
& \nu y_1 y_1'\, \nu y_0 y_0'\, (P_1' \mid y_0'(b').\, y\langle b'\rangle .\, \mathsf{merge}_y(i.i\langle y_0'\rangle .\, i \leftrightarrow y_1')) \\
& \vdots \\
& \downarrow l_n \qquad\qquad \text{By rule } \textsc{Res} \text{ if } P_{n-1}' \xrightarrow{l_n} P_n' \text{ and } y_0, y_1 \notin \mathrm{N}(l_n) \\
X_2 = {} & \nu y_1 y_1'\, \nu y_0 y_0'\, (P_n' \mid y_0'(b').\, y\langle b'\rangle .\, \mathsf{merge}_y(i.i\langle y_0'\rangle .\, i \leftrightarrow y_1')) \\
& \downarrow\tau \qquad\qquad \text{By } \textsc{Tensor-Par} \text{ for } y_0, y_0',\ P_n' \xrightarrow{y_0[b']} P'' \\
& \nu y_1 y_1'\, \nu y_0 y_0'\, \nu bb'\, (P'' \mid y\langle b'\rangle .\, \mathsf{merge}_y(i.i\langle y_0'\rangle .\, i \leftrightarrow y_1')) \\
& \big\downarrow y[b''] \\
& \downarrow\tau \qquad\qquad \text{By } \textsc{Link} \text{ for } b', b'' \\
X_3 = {} & \nu y_1 y_1'\, \nu y_0 y_0'\, (P'' \mid \mathsf{merge}_y(i.i\langle y_0'\rangle .\, i \leftrightarrow y_1')) \\
& \big\downarrow \diamond y^{\mathsf{j}}(a') \\
X_4 = {} & \nu y_1 y_1'\, \nu y_0 y_0'\, (P'' \mid \nu iz''\, \nu iz\, (Q^{\mathsf{j}} \mid i\langle y_0'\rangle .\, i \leftrightarrow y_1' \mid i'(b').\, y\langle b\rangle .\, \mathsf{merge}_y(i.i \leftrightarrow y_1')))
\end{aligned}
$$

Observe that the processes Z_4 and X_5 differ only for the arrangement of restrictions and thus $Z_4 \sim X_4$ since $\nu xy\, (P \mid Q) \sim (\nu xy\, P \mid Q)$ for any P and Q such that $x, y \notin \textsc{Fn}(Q)$. All transitions that lead us from Z to Z_4 have a matching transition in (saturation of) the trace from X to X_4 and vice versa. In fact, following the same reasoning we can build the trace for Z starting from the one for X. Other cases can only differ because of interleaving of concurrent transitions and of the action on y selected for Z_3, which is immaterial. $\square$

Proof of proposition 20. Let $S := \Diamonddot^{\Theta^\perp} D^\perp$ and we derive

$$\dfrac{\begin{array}{c} x \leftrightarrow i \vdash x : S^\perp, i : S \\ \mathsf{absorb}_{a'z'z} \vdash z : S^\perp, a' : A^\perp, z' : B^\perp \otimes S \qquad \dfrac{T \vdash x : D, y : C}{\boxdot x[].\, T \vdash x : S^\perp, y : C} \end{array}}{\mathsf{fmap}_{xy}(T) := \Diamonddot y\{i.\, x \leftrightarrow i, za'z'.\, \mathsf{absorb}_{a'z'z}, x.\, \boxdot x[].\, T\} \vdash x : \boxdot^{\Theta} D, y : \Diamonddot^{\Theta^\perp} C}$$

Let $S := \Diamonddot^{\Theta}(C \parr D) \otimes C^\perp$ and we derive

$$\dfrac{\begin{array}{c} \dfrac{\begin{array}{c} P := i \leftrightarrow i' \vdash i' : S^\perp, i : S \\ \vdots \\ Q \qquad \dfrac{\overline{y'[f'].\,(f \leftrightarrow f' \mid y \leftrightarrow y') \vdash f : C, y' : C^\perp \otimes D^\perp, y : D}}{R := f(y).\, \boxdot y'[].\, \cdots \vdash f : S^\perp, y : D}\,\textsc{BoW} \end{array}}{\Diamonddot y\{i.\, P, aa'z'.\, Q, f.\, R\} \vdash i' : S^\perp, y : \Diamonddot^{\Theta} D}\,\textsc{Di} \\ i[y].\,(x \leftrightarrow y \mid i \leftrightarrow c) \vdash i : S, x : \boxdot^{\Theta^\perp} C^\perp \otimes D^\perp, c : C \end{array}}{\mathsf{lift}_{xcy} := \nu i i' \; \cdots \vdash c : C, y : \Diamonddot^{\Theta} D, x : (\Diamonddot^{\Theta}(C \parr D))^\perp}\,\textsc{Cut}$$

where Q is

$$\dfrac{\begin{array}{c} z \leftrightarrow z' \vdash z : C, z' : C^\perp \\ \dfrac{\dfrac{\mathsf{i} : (A : B) \in \Theta}{\mathsf{i} : (A^\perp : B^\perp) \in \Theta^\perp}}{absorb_{a'yx} \vdash x : \boxdot^{\Theta^\perp}(C^\perp \otimes D^\perp), a' : A, y : B \otimes \Diamonddot^{\Theta}(C \parr D)}\,\text{PROPOSITION 29} \end{array}}{z(x).\, z'[y].\,(z \leftrightarrow z' \mid absorb_{a'yx}) \vdash z : S^\perp, a' : A, z' : B \otimes S}\,\textsc{Tensor}$$

Let $S := \Diamonddot^{\Theta^\perp} C^\perp$ and we derive

$$\dfrac{\begin{array}{c} i \leftrightarrow i' \vdash i' : S^\perp, i : S \\ \mathsf{absorb}_{a'z'z} \vdash z : S^\perp, a' : A^\perp, z' : B^\perp \otimes S \qquad f \leftrightarrow y \vdash f : S^\perp, y : S \end{array}}{\mathsf{flat}_{yi'} := \Diamonddot y\{i.\, P, za'z'.\, Q, f.\, R\} \vdash i' : S^\perp, y : \Diamonddot^{\Theta^\perp} S}$$

□

The following result is dual to lift; they are not used in our paper but still worth mentioning.

Proposition 31.

$$\boxdot^{\Theta}(C \parr D) \multimap (C \parr \boxdot^{\Theta} D) \qquad\qquad C \otimes \Diamonddot^{\Theta} D \multimap \Diamonddot^{\Theta}(C \otimes D)$$

Proof. we derive:

$$\dfrac{\dfrac{}{C, \boxdot^{\Theta} D, C^\perp \otimes D^\perp}\,\textsc{Ax} \qquad \mathcal{D} \qquad \dfrac{\dfrac{}{C, D, C^\perp \otimes D^\perp}\,\textsc{Ax}}{C, \boxdot^{\Theta} D, C^\perp \otimes D^\perp}\,\textsc{BoW}}{(\boxdot^{\Theta}(C \parr D))^\perp, C, \boxdot^{\Theta} D}\,\textsc{Di}$$

where $\mathcal{D}$ is

$$\dfrac{C, C^{\perp} \qquad \dfrac{\mathsf{i} : (A : B) \in \Theta}{\boxdot^{\Theta} D, A^{\perp}, B^{\perp} \otimes \Diamond\!\!\!\!\cdot\;^{\Theta^{\perp}} D^{\perp}}\,\text{PROPOSITION 29}}{C, \boxdot^{\Theta} D, A^{\perp}, B^{\perp} \otimes C^{\perp} \otimes \Diamond\!\!\!\!\cdot\;^{\Theta^{\perp}} D^{\perp}}\,\textsc{Tensor}$$

□

3.C Omitted metatheoretic proofs

We recall some notions from Montesi and Peressotti [2021], with our own simplications.

- Recall that lts_d, lts_p and lts_e refer respectively to the LTS of the derivations Der, processes $Proc$ and (hyper)environments Env.
- Recall that there is projection $proc : Der \to Proc$ and $env : Der \to Env$. We extend $proc$ and env to the powersets of Der; e.g., $proc(lts_d(\mathcal{D}, l)) := \{proc(\mathcal{E}) \mid \mathcal{E} \in lts_d(\mathcal{D}, l)\}$.
- We extend $proc$ and env to the typing rules. e.g., $proc(\textsc{BoCW})$ refers to projection of BoCW in processes (ltd_p), and $env(\textsc{BoCW})$ refers to projection in environments (lts_e).
- We use processes terms to represent derivations. For example, $\boxdot x[x_0, x_1]\, \mathcal{D}$ refers to the derivation of applying BoC to $\mathcal{D}$.

Proof of theorem 22. We need to show that $\{(proc(\mathcal{D}), \mathcal{D}) \mid \mathcal{D} \in Der\}$ is a strong bisimulation for lts_p and lts_d, which is to prove $proc(lts_d(\mathcal{D}, l)) = lts_p(proc(\mathcal{D}), l)$ for any $\mathcal{D}, l \in Der \times lbl$. Prove this by induction on the structure of $\mathcal{D}$; we simply extend Montesi and Peressotti [2021] with extra cases.

- If $\mathcal{D}$ is HMix0 or Ax, it is trivial.
- If $\mathcal{D}$ is HMix, then $proc\mathcal{D}$ is $proc(D_0) \mid proc(D_1)$, then we have

$$\begin{aligned}
& proc(lts_d(\mathcal{D}, l)) & \\
=& proc(lts_d(\mathcal{D}_0, l)) & \text{(by }\textsc{Par0}\text{)}\\
\cup& proc(lts_d(\mathcal{D}_1, l)) & \text{(by }\textsc{Par1}\text{)}\\
\cup& proc(lts_d(\mathcal{D}_0, l_0)) \mid proc(lts_d(\mathcal{D}_1, l_1)) & (l = l_0 \mid l_1\text{; by }\textsc{Syn}\text{)}\\
=& lts_p(proc(\mathcal{D}_0), l) & \text{(By I.H.)}\\
\cup& lts_p(proc(\mathcal{D}_1), l) & \text{(By I.H.)}\\
\cup& lts_p(proc(\mathcal{D}_0), l_0) \mid lts_p(proc(\mathcal{D}_1), l_1) & \text{(By I.H.)}\\
=& lts_p(proc(\mathcal{D}), l) & \text{(By } proc(\textsc{Par0}, \textsc{Par1}, \textsc{Syn})\text{)}
\end{aligned}$$

- If $\mathcal{D}$ is some action rule R and therefore $proc(\mathcal{D}) = \pi.proc(\mathcal{D}')$ where π is some prefix. Then the only non-trivial l is π and it follows easily.
- If $\mathcal{D}$ is BoA on $\mathcal{E}$, then $proc(\mathcal{D})$ is $\boxdot x^{\mathsf{i}}[a].\,proc(\mathcal{E})$. The only non-trivial l is $\boxdot x^{\mathsf{i}}[a]$ and

$$\begin{aligned} & proc(lts_d(\mathcal{D}, l)) & \\ =& proc(\{\mathcal{E}\}) & \text{(by BoAA)} \\ =& lts_p(proc(\mathcal{D}), l) & \text{(by } proc\text{(BoAA))} \end{aligned}$$

- If $\mathcal{D}$ is BoW on $\mathcal{E}$, then $proc(\mathcal{D})$ is $\boxdot x[].\,proc(\mathcal{E})$. The only non-trivial l is $\boxdot x[]$ and similar to last case.
- If $\mathcal{D}$ is BoR, the only non-trivial l is τ, and similar to last case.
- If $\mathcal{D}$ is Di, the only non-trivial l is either $\Diamonddot x()$ or $\Diamonddot x^{\mathsf{i}}(a)$; both cases are similar to last case.
- If $\mathcal{D}$ is BoC on $\mathcal{E}$, then $proc(\mathcal{D})$ is $\boxdot x[x_0, x_1]\,proc(\mathcal{E})$. The non-trivial l could be:
  - if l is $l' \mid \boxdot x[]$, then

$$\begin{aligned} & proc(lts_d(\mathcal{D}, l)) & \\ =& \nu x_0 y_0\, \nu x_1 y_1\, (Q \mid proc(lts_d(\mathcal{E}, l' \mid \boxdot x_0[] \mid \boxdot x_1[]))) & \text{(by BoCW)} \\ =& \nu x_0 y_0\, \nu x_1 y_1\, (Q \mid lts_p(proc(\mathcal{E}), l' \mid \boxdot x_0[] \mid \boxdot x_1[])) & \text{(by I.H.)} \\ =& lts_p(proc(\mathcal{D}), l) & \text{(by } proc\text{(BoCW))} \end{aligned}$$

  - if l is $l' \mid \boxdot x^{\mathsf{i}}[a]$, then

$$\begin{aligned} & proc(lts_d(\mathcal{D}, l)) \\ =& \nu x_0 y_0\, \nu x_1 y_1\, (Q \mid proc(lts_d(\mathcal{E}, l' \mid \boxdot x_0{}^{\mathsf{i}}[a]) \cup lts_d(\mathcal{E}, l' \mid \boxdot x_1{}^{\mathsf{i}}[a]))) \\ & \text{(by BoCA0,BoCA1)} \\ =& \nu x_0 y_0\, \nu x_1 y_1\, (Q \mid proc(lts_d(\mathcal{E}, l' \mid \boxdot x_0{}^{\mathsf{i}}[a])) \cup proc(lts_d(\mathcal{E}, l' \mid \boxdot x_1{}^{\mathsf{i}}[a]))) \\ & \text{(by extended def. of } proc\text{)} \\ =& \nu x_0 y_0\, \nu x_1 y_1\, (Q \mid lts_p(proc(\mathcal{E}), l' \mid \boxdot x_0{}^{\mathsf{i}}[a]) \cup lts_p(proc(\mathcal{E}), l' \mid \boxdot x_1{}^{\mathsf{i}}[a])) \\ & \text{(by I.H.)} \\ =& lts_p(proc(\mathcal{D}), l) \qquad \text{(by } proc\text{(BoCA0)}, proc(BoCA1)) \end{aligned}$$

  - Otherwise, we have

$$\begin{aligned} & proc(lts_d(\mathcal{D}, l)) & \\ =& \boxdot x[x_0, x_1]\, proc(lts_d(\mathcal{E}, l)) & \text{(by BoCP)} \\ =& \boxdot x[x_0, x_1]\, lts_p(proc(\mathcal{E}), l) & \text{(by I.H.)} \\ =& lts_p(proc(\mathcal{D}), l) & \text{(by } proc\text{(BoCP))} \end{aligned}$$

- If $\mathcal{D}$ is Cut on $\mathcal{E}$, the non-trivial l are:
  - if l is not τ, we should have

$$\begin{aligned}
&proc(lts_d(\mathcal{D}, l)) & \\
=&\nu xy\, proc(lts_d(\mathcal{E}, l)) & \text{(by Res)} \\
=&\nu xy\, lts_p(proc(\mathcal{E}), l) & \text{(by I.H.)} \\
=&lts_p(proc(\mathcal{D}), l) & \text{(by } proc(\text{Res}))
\end{aligned}$$

  - Othewise l is τ, then

$$\begin{aligned}
&proc(lts_d(\mathcal{D}, \tau)) \\
= \nu xy.proc(&lts_d(\mathcal{E}, x() \mid y[]) \cup \\
&lts_d(\mathcal{E}, x(x') \mid y[y']) \cup \\
&\ldots \\
&lts_d(\mathcal{E}, \boxdot x[] \mid \Diamonddot y()) \cup \\
&lts_d(\mathcal{E}, \boxdot x^{\mathsf{i}}[a] \mid \Diamonddot y^{\mathsf{i}}(a'))) \\
&\qquad\qquad\qquad\qquad \text{(by all communication rules)} \\
= \nu xy.(&lts_p(proc(\mathcal{E}), x() \mid y[]) \cup \\
&lts_p(proc(\mathcal{E}), x(x') \mid y[y']) \cup \\
&\ldots \\
&lts_p(proc(\mathcal{E}), \boxdot x[] \mid \Diamonddot y()) \cup \\
&lts_p(proc(\mathcal{E}), \boxdot x^{\mathsf{i}}[a] \mid \Diamonddot y^{\mathsf{i}}(a'))) \qquad \text{(by I.H.)} \\
=&lts_p(proc(\mathcal{D}), \tau) \qquad \text{(by } proc(allcommrules))
\end{aligned}$$

□

Theorem 32 (Non-interference)**.** *If $\mathcal{D} \xrightarrow{l_0}$ and $\mathcal{D} \xrightarrow{l_1}$, where $l_0 \# l_1$, then $\mathcal{D} \xrightarrow{l_0 | l_1}$.*

Proof. By induction on the derivation of $\mathcal{D}$.

- If $\mathcal{D}$ is HMix0, it is trivial.
- If $\mathcal{D}$ is HMix on $\mathcal{D}_0$ and $\mathcal{D}_1$, then we take cases on $\mathcal{D} \xrightarrow{l_0}$ and $\mathcal{D} \xrightarrow{l_1}$.
  - If both are by Par0, then we invoke I.H. on $\mathcal{D}_0$, then apply Par0.
  - If one is by Par0 and the other by Par1, then we invoke Syn.
  - If $\mathcal{D} \xrightarrow{l_0}$ is by Par0 which means $\mathcal{D}_0 \xrightarrow{l_0}$, and $\mathcal{D} \xrightarrow{l_1}$ is by Syn which means $\mathcal{D}_0 \xrightarrow{l_{10}}$ and $\mathcal{D}_1 \xrightarrow{l_{11}}$, then we first invoke I.H. on $\mathcal{D}_0$, then invoke Syn.

  - If both are by SYN, then we will invoke I.H. on $\mathcal{D}_0$ and $\mathcal{D}_1$ respectively, then invoke SYN.
  - All other cases are similar.

- If $\mathcal{D}$ is by AX or some action rules, it vacuously holds because it cannot be the case that $l_0 \# l_1$.

- If $\mathcal{D}$ is by CUT, that means it is of the shape $\nu xy\, \mathcal{D}'$. We take cases on $\mathcal{D} \xrightarrow{l_0}$ and $\mathcal{D} \xrightarrow{l_1}$

  - If $\mathcal{D} \xrightarrow{l_0}$ is by some communication rule, which means l_0 is τ, that becomes trivial as $\tau \mid l_1$ is l_1.
  - If both are by RES, that means $\mathcal{D}' \xrightarrow{l_0}$ and $\mathcal{D}' \xrightarrow{l_1}$. Invoke I.H. and apply RES.

- If $\mathcal{D}$ is BOW or BOA or DI on $\mathcal{E}$, trivial because it cannot be the case that $l_0 \# l_1$.

- If $\mathcal{D}$ is BOR, trivial because can only be $\tau \notin \mathit{Act}$.

- If $\mathcal{D}$ is BOC to $\mathcal{E}$, then $\mathit{proc}(\mathcal{D}) = \boxdot x[x_0, x_1]\, \mathit{proc}(\mathcal{E})$. Several cases:

  - If both l_0, l_1 are by BOCP, then $\mathcal{E} \xrightarrow{l_0}$ and $\mathcal{E} \xrightarrow{l_1}$, then by I.H. $\mathcal{E} \xrightarrow{l_0|l_1}$, then apply BOCP and $D \xrightarrow{l_0|l_1}$
  - If l_0 is by BOCP and l_1 is by BOCA0. Say l_1 is $l' \mid \boxdot x^{\mathrm{i}}[a]$, then $\mathcal{E} \xrightarrow{l_0}$ and $\mathcal{E} \xrightarrow{l'|\boxdot x_0{}^{\mathrm{i}}[a]}$, then by I.H. $\mathcal{E} \xrightarrow{l_0|l'|\boxdot x_0{}^{\mathrm{i}}[a]}$, then apply BOCA0 and $\mathcal{D} \xrightarrow{l_0|l'|\boxdot x^{\mathrm{i}}[a]}$.
  - If l_0 is by BOCP and l_1 is by BOCW. Say l_1 is $l' \mid \boxdot x[]$, then $\mathcal{E} \xrightarrow{l_0}$ and $\mathcal{E} \xrightarrow{l'|\boxdot x_0[]|\boxdot x_1[]}$. Then by I.H. $\mathcal{E} \xrightarrow{l_0|l'|\boxdot x_0[]|\boxdot x_1[]}$, then apply BOCW and $\mathcal{D} \xrightarrow{l_0|l'|\boxdot x[]}$.
  - All other cases are similar or impossible.

□

Proof of theorem 23. We only need to prove $\mathit{env}(\mathit{lts}_d(\mathcal{D}, l)) \subset \mathit{lts}_e(\mathit{env}(\mathcal{D}), l)$ for any derivation $\mathcal{D}$ and label l. Prove by induction on $\mathcal{D}$.

- If $\mathcal{D}$ is by HMIX0, it is trivial.

- If $\mathcal{D}$ is by HMix on $\mathcal{D}_0$ and $\mathcal{D}_1$, and thus $env(\mathcal{D}) = env(\mathcal{D}_0) \mid env(\mathcal{D}_1)$. We consider the members of $env(lts_d(\mathcal{D}, l))$; to do that, we consider how $\mathcal{D} \xrightarrow{l}$ is derived.

  - if by Par0, then for any

  $$\begin{aligned} & env(\mathcal{D}'_0) \mid env(\mathcal{D}_1) \in env(lts_d(\mathcal{D}, l)) & \\ & \qquad env(\mathcal{D}'_0) \in env(lts_d(\mathcal{D}_0, l)) & \text{(by Par0)} \\ & \qquad env(\mathcal{D}'_0) \in lts_e(env(\mathcal{D}_0), l) & \text{(by I.H.)} \\ & env(\mathcal{D}'_0) \mid env(\mathcal{D}_1) \in lts_e(env(\mathcal{D}_0) \mid env(\mathcal{D}_1), l) & \text{(by } env(\text{Par0})) \end{aligned}$$

  - the case for Par1 is similar.
  - If by Syn, then $l = l_0 \mid l_1$ and for any

  $$\begin{aligned} & env(\mathcal{D}'_0) \mid env(\mathcal{D}'_1) \in env(lts_d(\mathcal{D}, l)) \\ & env(\mathcal{D}'_0) \in env(lts_d(\mathcal{D}_0, l_0)) \text{ and } env(\mathcal{D}'_1) \in env(lts_d(\mathcal{D}_1, l_1)) \\ & \hfill \text{(by Syn)} \\ & env(\mathcal{D}'_0) \in lts_e(env(\mathcal{D}_0), l_0) \text{ and } env(\mathcal{D}'_1) \in lts_e(env(\mathcal{D}_1), l_1) \\ & \hfill \text{(by I.H.)} \\ & env(\mathcal{D}'_0) \mid env(\mathcal{D}'_1) \in lts_e(env(\mathcal{D}_0) \mid env(\mathcal{D}_1), l) \\ & \hfill \text{(by } env(\text{Syn})) \end{aligned}$$

- If $\mathcal{D}$ is BoA on $\mathcal{E}$, therefore $env(\mathcal{E})$ is $\Delta, a : A \mid \Gamma, b : B, x : \boxdot^{\Theta} C$ and $env(\mathcal{D})$ is $\Delta, \Gamma, x : \boxdot^{\Theta} C$ where $\mathsf{i} : (A : B) \in \Theta$. The only non-trivial l is $\boxdot x^{\mathsf{i}}[a]$ and

  $$\begin{aligned} & env(lts_d(\mathcal{D}, l)) & \\ = & env(\mathcal{E}) & \text{(by BoAA)} \\ \in & lts_e(env(\mathcal{D}), l) & \text{(by } env(\text{BoAA})) \end{aligned}$$

- If $\mathcal{D}$ is BoW or Di or BoR similar.

- If $\mathcal{D}$ is by Ax or other action rules, similar.

- If $\mathcal{D}$ is apply BoC on $\mathcal{E}$, then $env(\mathcal{E})$ is $\mathcal{G} \mid \Gamma, x_0 : \boxdot^{\Theta} C \mid \Delta, x_1 : \boxdot^{\Theta} D$ and $env(\mathcal{D})$ is $\mathcal{G} \mid \Gamma, \Delta, x : \boxdot^{\Theta} (C \otimes D)$. Take cases:

  - If $x \notin \text{Fn}(l)$, then for any

  $$\begin{aligned} \mathcal{G}' \mid \Gamma', \Delta', x : \boxdot^{\Theta} (C \otimes D) & \in env(lts_d(\mathcal{D}, l)) & \\ \mathcal{G}' \mid \Gamma', x_0 : \boxdot^{\Theta} C \mid \Delta', x_1 : \boxdot^{\Theta} D & \in env(lts_d(\mathcal{E}, l)) & \text{(by BoCP)} \\ & \subset lts_e(env(\mathcal{E}), l) & \text{(by I.H.)} \\ \mathcal{G}' \mid \Gamma', \Delta', x : \boxdot^{\Theta} (C \otimes D) & \in lts_e(env(\mathcal{D}), l) & \\ & & \text{(by } env(\text{BoCP})) \end{aligned}$$

- If l is $l' \mid \boxdot x[]$, then for any

$$\begin{aligned}
\mathcal{G}' \mid \Gamma, \Delta, x : C \otimes D &\in env(lts_d(\mathcal{D}, l' \mid \boxdot x[])) \\
\mathcal{G}' \mid \Gamma, x_0 : C \mid \Delta, x_1 : D &\in env(lts_d(\mathcal{E}, l' \mid \boxdot x_0[] \mid \boxdot x_1[])) \\
&\qquad \text{(by BoCW)} \\
&\subset lts_e(env(\mathcal{E}), l' \mid \boxdot x_0[] \mid \boxdot x_1[]) \\
&\qquad \text{(by I.H.)} \\
\mathcal{G}' \mid \Gamma, \Delta, x : C \otimes D &\in lts_e(env(\mathcal{D}), l' \mid \boxdot x[]) \\
&\qquad \text{(by env(BoCW))}
\end{aligned}$$

- If l is of shape $l' \mid \boxdot x^{\mathsf{i}}[a]$, then for any

$$\begin{aligned}
\mathcal{G} \mid \Gamma_0, a : A \mid \Gamma_1, \Delta, x : B \parr \boxdot^{\Theta}(C \otimes D) &\in env(lts_d(\mathcal{D}, l' \mid \boxdot x^{\mathsf{i}}[a])) \\
\mathcal{G} \mid \Gamma_0, a : A \mid \Gamma_1, x_0 : B \parr \boxdot^{\Theta} C \mid \Delta, x_1 : \boxdot^{\Theta} D &\in env(lts_d(\mathcal{E}, l' \mid \boxdot x_0^{\mathsf{i}}[a])) \\
&\qquad \text{(by BoCA0)} \\
&\subset lts_e(env(\mathcal{E}), l' \mid \boxdot x_0^{\mathsf{i}}[a]) \\
&\qquad \text{(by I.H.)} \\
\mathcal{G} \mid \Gamma_0, a : A \mid \Gamma_1, \Delta, x : B \parr \boxdot^{\Theta}(C \otimes D) &\in lts_e(env(\mathcal{D}), l) \\
&\qquad \text{(by } env\text{(BoCA0))}
\end{aligned}$$

  There is also

$$\mathcal{G} \mid \Delta_0, a : A \mid \Delta_1, \Gamma, x : B \parr \boxdot^{\Theta}(C \otimes D) \in env(lts_d(\mathcal{D}, l' \mid \boxdot x^{\mathsf{i}}[a]))$$

  but symmetric and thus omitted.

- other cases are similar or trivial.

- If $\mathcal{D}$ is Cut on $\mathcal{E}$, then $env(\mathcal{E})$ is $\mathcal{G} \mid \Gamma, x : A \mid \Delta, y : A^{\perp}$ and $env(\mathcal{D})$ is $\mathcal{G} \mid \Gamma, \Delta$. If l is not τ, it is trivial; otherwise we have

$$\begin{aligned}
& env(lts_d(\mathcal{D}, \tau)) \\
&= env(lts_d(\mathcal{E}, \boxdot x^{\mathsf{i}}[a] \mid \Diamonddot y^{\mathsf{i}}(a')) \cup \\
&\qquad lts_d(\mathcal{E}, \boxdot x[] \mid \Diamonddot y()) \cup \\
&\qquad \cdots) \qquad \text{(by all comm rules)} \\
&= env(lts_d(\mathcal{E}, \boxdot x^{\mathsf{i}}[a] \mid \Diamonddot y^{\mathsf{i}}(a'))) \cup \\
&\qquad env(lts_d(\mathcal{E}, \boxdot x[] \mid \Diamonddot y())) \\
&\qquad \cdots \qquad \text{(by extended def of } env\text{)} \\
&\subset lts_e(env(E), \boxdot x^{\mathsf{i}}[a] \mid \Diamonddot y^{\mathsf{i}}(a')) \cup \\
&\qquad lts_e(env(E), \boxdot x[] \mid \Diamonddot x()) \cup \\
&\qquad \cdots \qquad \text{(by I.H.)} \\
&= lts_e(env(D), \tau) \qquad \text{(by } env\text{(all comm rules))}
\end{aligned}$$

□

Lemma 33 (External choice of coemitter)**.** *If $P \vdash \mathcal{G} \mid \Gamma, y : \Diamonddot^{i,j,\cdots} C$, then the following are equivalent:*

- $P \xrightarrow{\Diamonddot y()}$
- $P \xrightarrow{\Diamonddot y^{i}(a')}$
- $P \xrightarrow{\Diamonddot y^{j}(a')}$
- $\cdots$

Proof. The only ways those labels can be emitted are by DiW or by DiA; and the two have the same premises. Formally, one can prove this by induction on $P \vdash \mathcal{G} \mid \Gamma, y : \Diamonddot^{i,j,\cdots} C$. □

Proof of theorem 25. We strengthen the statement to the following. Let $P \vdash \Gamma_0 \mid \cdots \mid \Gamma_{n-1}$. For every $i < n$, we have $P \xrightarrow{l_i}$ where either $l_i = \tau$ or $\mathrm{Fn}(l_i) \cap \mathrm{N}(\Gamma_i) \neq \emptyset$. This is so we can prove by induction on the derivaiton of $P \vdash \mathcal{G}$.

HMix0 Trivial.

HMix then P is $P_0 \mid P_1$ where

$$\begin{aligned} P_0 &\vdash \Gamma_0 \mid \cdots \mid \Gamma_{m-1} \\ P_1 &\vdash \Gamma_m \mid \cdots \mid \Gamma_{n-1} \end{aligned}$$

For $i < m$ we apply I.H. on P_0, and apply Par0. For $m \leq i < n$ we apply I.H. on P_1 and apply Par1.

BoC then P is $\boxdot x[x_0, x_1]\, P'$ where

$$\begin{aligned} P' &\vdash \Gamma_0 \mid \cdots \mid \Gamma_{n-1}, x_0 : \boxdot^{\Theta} C \mid \Gamma_n, x_1 : \boxdot^{\Theta} D \\ P &\vdash \Gamma_0 \mid \cdots \mid \Gamma_{n-1}, \Gamma_n, x : \boxdot^{\Theta} (C \otimes D) \end{aligned}$$

Apply I.H. on P' and we get a series of labels $l_0, \cdots, l_n$. For $i < n-1$, we apply BoCP on l_i and we are done. For $i = n-1$, we look at l_{n-1} and l_n:

- If $\mathrm{Fn}(l_{n-1}) \neq x_0$, then we apply BoCP.
- If l_{n-1} is $\boxdot x_0{}^{\mathrm{i}}[a]$, then we apply BoCA0.
- If $\mathrm{Fn}(l_n) \neq x_1$, then apply BoCP.
- If l_n is $\boxdot x_1{}^{\mathrm{i}}[a]$, then we apply BoCA1.

- Otherwise it must be that l_{n-1} is $\boxdot x_0[]$ and l_n is $\boxdot x_1[]$, and we first invoke theorem 32 to get $P' \xrightarrow{\boxdot x_0[] | \boxdot x_1[]}$ and then apply BoCW to get $P \xrightarrow{\boxdot x[]}$.

Cut Then P is $\nu xy\, P'$, where

$$\begin{aligned} P' &\vdash \Gamma_0 \mid \cdots \mid \Gamma_{n-1}, x : A \mid \Gamma_n, y : A^\perp \\ P &\vdash \Gamma_0 \mid \cdots \mid \Gamma_{n-1}, \Gamma_n \end{aligned}$$

Apply I.H. and we get a series of labels $l_0, \cdots, l_n$. For $i < n-1$, we apply Res on l_i and we are done. For $i = n-1$, we look at l_{n-1} and l_n:

- If $\textsc{Fn}(l_{n-1}) \neq x$, then we apply Res.
- If $\textsc{Fn}(l_n) \neq y$, then we apply Res.
- Otherwise we must have $\textsc{Fn}(l_{n-1}) = x$ and $\textsc{Fn}(l_n) = y$. By theorem 23 we know the the two labels must be dual. In particular, if A or $A^\perp$ is $\Diamonddot$ one has to invoke lemma 33 to ensure that $\Diamonddot$ emits actions matching with $\boxdot$.

o.w. then $n = 1$ and we have the corresponding action rule giving transition.

□

Theorem 34 (Partial Diamond Property)**.** *lts_d with restriction on BoCA0 and BoCA1 enjoys the diamond property. To be precise: if $\mathcal{D} \xrightarrow[=]{l_0} \mathcal{E}$ (left premise) and $\mathcal{D} \xrightarrow{l_1} \mathcal{F}$ (right premise) where $l_0 \# l_1$ and not $\mathcal{E} =_\alpha \mathcal{F}$, then there exists $\mathcal{G}$ such that $\mathcal{E} \xrightarrow{l_1} \mathcal{G}$ and $\mathcal{F} \xrightarrow[=]{l_0} \mathcal{G}$. This generalizes the diamond property in Montesi and Peressotti [2021].*

Proof of theorem 34. By induction on the derivation of $\mathcal{D}$. We first notice that in most cases, $l_0 \# l_1$ is false, and therefore trivial. We only consider the following cases.

HMix $\mathcal{D}$ is $D_0 \mid D_1$. We take cases

- If $\mathcal{D} \xrightarrow[=]{l_0} \mathcal{E}$ is Par0 on $\mathcal{D}_0 \xrightarrow[=]{l_0} \mathcal{E}_0$, and $\mathcal{D} \xrightarrow{l_1} \mathcal{F}$ is Par0 on $\mathcal{D}_0 \xrightarrow{l_1} \mathcal{F}_0$, then apply I.H. we get some $\mathcal{G}_0$ so that $\mathcal{E}_0 \xrightarrow{l_1} \mathcal{G}_0$ and $\mathcal{F}_0 \xrightarrow[=]{l_0} \mathcal{G}_0$. Let $\mathcal{G} := \mathcal{G}_0 \mid \mathcal{D}_1$, and by Par0 we have $\mathcal{E} \xrightarrow{l_1} \mathcal{G}$ and $\mathcal{F} \xrightarrow[=]{l_0} \mathcal{G}$.
- If $\mathcal{D} \xrightarrow[=]{l_0} \mathcal{E}$ is Par0 on $\mathcal{D}_0 \xrightarrow[=]{l_0} \mathcal{E}_0$, and $\mathcal{D} \xrightarrow{l_1} \mathcal{F}$ is Par1 on $\mathcal{D}_1 \xrightarrow{l_1} \mathcal{F}_1$, let $\mathcal{G} := \mathcal{E}_0 \mid \mathcal{F}_1$, and (without I.H.) we will have $\mathcal{E} \xrightarrow{l_1} \mathcal{G}$ by Par1 and $\mathcal{F} \xrightarrow[=]{l_0} \mathcal{G}$ by Par0.

– If $\mathcal{D} \xrightarrow[=]{l_{00}|l_{01}} \mathcal{E}$ is Syn on $\mathcal{D}_0 \xrightarrow[=]{l_{00}} \mathcal{E}_0$ and $\mathcal{D}_1 \xrightarrow[=]{l_{01}} \mathcal{E}_1$, and $\mathcal{D} \xrightarrow{l_1} \mathcal{F}$ is Par0 on $\mathcal{D}_0 \xrightarrow{l_1} \mathcal{F}_0$. By I.H. we will have some $\mathcal{G}_0$ such that $\mathcal{E}_0 \xrightarrow{l_1} \mathcal{G}_0$ and $\mathcal{F}_0 \xrightarrow[=]{l_{00}} \mathcal{G}_0$. Let $\mathcal{G} := \mathcal{G}_0 \mid \mathcal{E}_1$, we should have $\mathcal{E} \xrightarrow{l_1} \mathcal{G}$ by Par0, and $\mathcal{F} \xrightarrow[=]{l_{00}|l_{01}} \mathcal{G}$ by Syn.

– All other cases are similar.

BoC $\mathcal{D}$ is $\boxdot x[x_0, x_1]\,\mathcal{D}'$. We take cases

– If $\mathcal{D} \xrightarrow[=]{l_0} \mathcal{E} = \boxdot x[x_0, x_1]\,\mathcal{E}'$ by BoCP and $\mathcal{D} \xrightarrow{l_1} \mathcal{F} = \boxdot x[x_0, x_1]\,\mathcal{F}'$ by BoCP. Then $\mathcal{D}' \xrightarrow[=]{l_0} \mathcal{E}'$ and $\mathcal{D}' \xrightarrow{l_1} \mathcal{F}'$, and by I.H. we have some $\mathcal{G}'$ such that $\mathcal{E}' \xrightarrow{l_1} \mathcal{G}'$ and $\mathcal{F}' \xrightarrow[=]{l_0} \mathcal{G}'$. Let $\mathcal{G} := \boxdot x[x_0, x_1]\,\mathcal{G}'$ and we have $\mathcal{E} \xrightarrow{l_1} \mathcal{G}$ and $\mathcal{F} \xrightarrow[=]{l_0} \mathcal{G}$ both by BoCP.

– If $\mathcal{D} \xrightarrow[=]{l_0} \mathcal{E} = \boxdot x[x_0, x_1]\,\mathcal{E}'$ by BoCP and $\mathcal{D} \xrightarrow{l_1'|\boxdot x[]} \mathcal{F} = \nu x_0 y_0\, \nu x_1 y_1\, (Q \mid \mathcal{F}')$ by BoCW. Then $\mathcal{D}' \xrightarrow[=]{l_0} \mathcal{E}'$ and $\mathcal{D}' \xrightarrow{l_1'|\boxdot x_0[]|\boxdot x_1[]} \mathcal{F}'$. By I.H. there exist some $\mathcal{G}'$ such that $\mathcal{E}' \xrightarrow{l_1'|\boxdot x_0[]|\boxdot x_1[]} \mathcal{G}'$ and $\mathcal{F}' \xrightarrow[=]{l_0} \mathcal{G}'$. Let $\mathcal{G} := \nu x_0 y_0\, \nu x_1 y_1\, (Q \mid \mathcal{G}')$ and we have $\mathcal{E} \xrightarrow{l_1'|\boxdot x[]} \mathcal{G}$ by BoCW and $\mathcal{F} \xrightarrow[=]{l_0} \mathcal{G}$ by Par1 and Res.

– If $\mathcal{D} \xrightarrow[=]{l_0} \mathcal{E} = \boxdot x[x_0, x_1]\,\mathcal{E}'$ by BoCP, and $\mathcal{D} \xrightarrow{l_1'|\boxdot x^{\mathsf{i}}[a]} \mathcal{F} = \nu x_0 y_0\, \nu x_1 y_1\, (Q \mid \mathcal{F}')$ by BoCA0, that gives $\mathcal{D}' \xrightarrow[=]{l_0} \mathcal{E}'$ and $\mathcal{D}' \xrightarrow{l_1'|\boxdot x_0{}^{\mathsf{i}}[a]} \mathcal{F}'$. By I.H. we get some $\mathcal{G}'$ such that $\mathcal{F}' \xrightarrow[=]{l_0} \mathcal{G}'$ and $\mathcal{E}' \xrightarrow{l_1'|\boxdot x_0{}^{\mathsf{i}}[a]} \mathcal{G}'$. Let $\mathcal{G} := \nu x_0 y_0\, \nu x_1 y_1\, (Q \mid \mathcal{G}')$ and we have $\mathcal{E} \xrightarrow{l_1'|\boxdot x^{\mathsf{i}}[a]} \mathcal{G}$ by BoCA0 and $\mathcal{F} \xrightarrow[=]{l_0} \mathcal{G}$ and Par1 and Res.

– All other cases are similar or trivial (in the sense that $l_0 \# l_1$ is false).

Cut $\mathcal{D}$ is $\nu xy\, \mathcal{D}'$. We take cases:

– Both premises are given by Res, then we have $\mathcal{D}' \xrightarrow[=]{l_0} \mathcal{E}'$ and $\mathcal{D}' \xrightarrow{l_1} \mathcal{F}'$ where $\mathcal{E} = \nu xy\, \mathcal{E}'$ and $\mathcal{F} = \nu xy\, \mathcal{F}'$. By I.H. we have some $\mathcal{G}'$ such that $\mathcal{F}' \xrightarrow[=]{l_0} \mathcal{G}'$ and $\mathcal{E}' \xrightarrow{l_1} \mathcal{G}'$. Define $\mathcal{G} := \nu xy\, \mathcal{G}'$ and by Res we have $\mathcal{F} \xrightarrow[=]{l_0} \mathcal{G}$ and $\mathcal{E} \xrightarrow{l_1} \mathcal{G}$.

- Right premise by Res and therefore $\mathcal{D}' \xrightarrow{l_1} \mathcal{F}'$ where $\mathcal{F} = \nu xy\, \mathcal{F}'$.

  Left premise by some communication rule R, and we have $\mathcal{D}' \xrightarrow[=]{l_0'} \mathcal{E}'$ for some l_0', where $\mathcal{E}'$ is $\mathcal{E}$ but with more or fewer cuts according to R. By I.H. we get some $\mathcal{G}'$ such that $\mathcal{F}' \xrightarrow[=]{l_0'} \mathcal{G}'$ and $\mathcal{E}' \xrightarrow{l_1} \mathcal{G}'$. Apply R on the first we get $\mathcal{F} \xrightarrow[=]{l_0} \mathcal{G}$, and apply Res several times on the second we get $\mathcal{E} \xrightarrow{l_1} \mathcal{G}$, where $\mathcal{G}$ is $\mathcal{G}'$ with more or fewer cuts according to R.
- Both premises are by some communication rule. This is impossible because the only way $\mathcal{E}$ could be different from $\mathcal{F}$ is if one side uses BoCA0 and the other BoCA1, which is preventd by the requirement that $D \xrightarrow[=]{l_0} E$.
- Other cases are similar.

□

Proof of lemma 27. The proof is similar to Montesi and Peressotti [2021]. Define $R :=\approx \cup\{(\mathcal{D},\mathcal{E}) \mid \mathcal{D} \xrightarrow[=]{\tau} \mathcal{E}\}$, we show R to be a bisimulation, and therefore part of $\approx$. For $(\mathcal{D},\mathcal{E}) \in R$ where $\mathcal{D} \xrightarrow[=]{\tau} \mathcal{E}$, we note:

- For any $\mathcal{D} \xrightarrow{l} \mathcal{D}'$. If $\mathcal{D}' = \mathcal{E}$, then l has to be τ, and we easily have $\mathcal{E} \xRightarrow{\tau} \mathcal{E}$ and $(\mathcal{E},\mathcal{E}) \in R$. Otherwise by theorem 34 there exists $\mathcal{E}'$ such that $\mathcal{E} \xrightarrow{l} \mathcal{E}'$ and $\mathcal{D}' \xrightarrow[=]{\tau} \mathcal{E}'$. That gives $\mathcal{E} \xRightarrow{l} \mathcal{E}'$ and $(\mathcal{D}',\mathcal{E}') \in R$.
- For any $\mathcal{E} \xrightarrow{l} \mathcal{E}'$, we have $\mathcal{D} \xRightarrow{l} \mathcal{E}'$ and $(\mathcal{E}',\mathcal{E}') \in R$.

□

3.D Omitted content of CEGV

The functional fragment of CEGV is given in fig. 3.D.1 and fig. 3.D.2. The session fragment is in fig. 3.D.3. Omitted rules of the effect fragment is in fig. 3.D.4.

Proposition 35. *$[\![C_U]\!]$ is positive.*

Proof. By simple induction on unlimited types C_U. □

Proposition 36. *$[\![-]\!]$ preserves duality. I.e., $[\![C_S]\!]^\perp = [\![\overline{C_S}]\!]$*

Proof. By simple induction on session types C_S. □

$$\left[\!\!\left[\begin{array}{c}\textsc{Contract}\\ \dfrac{\Gamma, x_0 : A_U, x_1 : A_U \vdash P : B}{\Gamma, x_0 : A_U \vdash P[x_0/x_1] : B}\end{array}\right]\!\!\right]_b :=$$

$$\dfrac{[\![P]\!]_b \vdash [\![\Gamma]\!]^\perp, x_0 : [\![A_U]\!]^\perp, x_1 : [\![A_U]\!]^\perp, b : [\![B]\!]}{x_0[\mathrm{DUP}](x_1).\ [\![P]\!]_b \vdash [\![\Gamma]\!]^\perp, x_0 : [\![A_U]\!]^\perp, b : [\![B]\!]}$$

$$\left[\!\!\left[\begin{array}{c}\textsc{Weaken}\\ \dfrac{\Gamma \vdash P : B}{\Gamma, x : A_U \vdash P : B}\end{array}\right]\!\!\right]_b := \dfrac{[\![P]\!]_b \vdash [\![\Gamma]\!]^\perp, b : [\![B]\!]}{x[\mathrm{DISP}].\ [\![P]\!]_b \vdash [\![\Gamma]\!]^\perp, x : [\![A_U]\!]^\perp, b : [\![B]\!]}$$

$$\left[\!\!\left[\begin{array}{c}\textsc{Tensor-I}\\ \dfrac{\Gamma \vdash P : C \qquad \Delta \vdash Q : D}{\Gamma, \Delta \vdash (P, Q) : C \otimes D}\end{array}\right]\!\!\right]_d :=$$

$$\dfrac{[\![P]\!]_c \vdash [\![\Gamma]\!]^\perp, c : [\![C]\!] \qquad [\![Q]\!]_d \vdash [\![\Delta]\!]^\perp, d : [\![D]\!]}{d[c].\ ([\![P]\!]_c \mid [\![Q]\!]_d) \vdash [\![\Gamma]\!]^\perp, [\![\Delta]\!]^\perp, d : [\![C]\!] \otimes [\![D]\!]}$$

$$\left[\!\!\left[\begin{array}{c}\textsc{Tensor-E}\\ \dfrac{\Gamma \vdash P : C \otimes D \qquad \Delta, c : C, d : D \vdash Q : E}{\Gamma, \Delta \vdash \mathsf{let}\ (c, d) = P\ \mathsf{in}\ Q : E}\end{array}\right]\!\!\right]_e :=$$

$$\dfrac{\begin{array}{c}[\![P]\!]_z \vdash [\![\Gamma]\!]^\perp, z : [\![C]\!] \otimes [\![D]\!]\\ \dfrac{[\![Q]\!]_e \vdash [\![\Delta]\!]^\perp, c : [\![C]\!]^\perp, d : [\![D]\!]^\perp, e : [\![E]\!]}{d(c).\ [\![Q]\!]_e \vdash [\![\Delta]\!]^\perp, d : [\![C]\!]^\perp \parr [\![D]\!]^\perp, e : [\![E]\!]}\end{array}}{\nu dz\ ([\![P]\!]_z \mid d(c).\ [\![Q]\!]_e) \vdash [\![\Gamma]\!]^\perp, [\![\Delta]\!]^\perp, e : [\![E]\!]} \qquad \left[\!\!\left[\begin{array}{c}\textsc{Star}\\ \dfrac{}{\vdash \star : \mathbf{1}}\end{array}\right]\!\!\right]_z := \dfrac{}{\mathbf{0}_z \vdash z : \mathbf{1}}$$

Figure 3.D.1: CEGV, functional fragment, Part one

$$\left\llbracket \dfrac{\textsc{Loli-I} \quad \Gamma, c : C \vdash P : D}{\Gamma \vdash \lambda c.\, P : C \multimap D} \right\rrbracket_d := \dfrac{\llbracket P \rrbracket_d \vdash \llbracket \Gamma \rrbracket^\perp, c : \llbracket C \rrbracket^\perp, d : \llbracket D \rrbracket}{d(c).\, \llbracket P \rrbracket_d \vdash \llbracket \Gamma \rrbracket^\perp, d : \llbracket C \rrbracket^\perp \parr \llbracket D \rrbracket}$$

$$\left\llbracket \dfrac{\textsc{Loli-E} \quad \Gamma \vdash P : C \multimap D \qquad \Delta \vdash Q : C}{\Gamma, \Delta \vdash P\, Q : D} \right\rrbracket_d :=$$

$$\dfrac{\llbracket P \rrbracket_z \vdash \llbracket \Gamma \rrbracket^\perp, z : \llbracket C \rrbracket^\perp \parr \llbracket D \rrbracket \quad \dfrac{\llbracket Q \rrbracket_c \vdash \llbracket \Delta \rrbracket^\perp, c : \llbracket C \rrbracket \qquad d \leftrightarrow d' \vdash d : \llbracket D \rrbracket, d' : \llbracket D \rrbracket^\perp}{d'[c].\, (\llbracket Q \rrbracket_c \mid d \leftrightarrow d') \vdash \llbracket \Delta \rrbracket^\perp, d : \llbracket D \rrbracket, d' : \llbracket C \rrbracket \otimes \llbracket D \rrbracket^\perp}}{\nu z d'\, (\llbracket P \rrbracket_z \mid d'[c].\, (\llbracket Q \rrbracket_c \mid d \leftrightarrow d')) \vdash \llbracket \Gamma \rrbracket^\perp, \llbracket \Delta \rrbracket^\perp, d : \llbracket D \rrbracket}$$

$$\left\llbracket \dfrac{\textsc{Arrow-I} \quad \Gamma_U \vdash P : C \multimap D}{\Gamma_U \vdash P : C \to D} \right\rrbracket_z := \dfrac{\llbracket P \rrbracket_z \vdash \llbracket \Gamma_U \rrbracket^\perp, z : \llbracket C \rrbracket^\perp \parr \llbracket D \rrbracket}{!z.\{\llbracket P \rrbracket_z\} \vdash \llbracket \Gamma_U \rrbracket^\perp, z : !(\llbracket C \rrbracket^\perp \parr \llbracket D \rrbracket)}$$

$$\left\llbracket \dfrac{\textsc{Arrow-E} \quad \Gamma \vdash P : C \to D}{\Gamma \vdash P : C \multimap D} \right\rrbracket_{w'} :=$$

$$\dfrac{\llbracket P \rrbracket_z \vdash \llbracket \Gamma \rrbracket^\perp, z : !(\llbracket C \rrbracket^\perp \parr \llbracket D \rrbracket) \quad \dfrac{w \leftrightarrow w' \vdash w : \llbracket C \rrbracket \otimes \llbracket D \rrbracket^\perp, w' : \llbracket C \rrbracket^\perp \parr \llbracket D \rrbracket}{w[\textsc{use}].\, w \leftrightarrow w' \vdash w : ?(\llbracket C \rrbracket \otimes \llbracket D \rrbracket^\perp), w' : \llbracket C \rrbracket^\perp \parr \llbracket D \rrbracket}}{\nu z w\, (\llbracket P \rrbracket_z \mid w[\textsc{use}].\, w \leftrightarrow w') \vdash \llbracket \Gamma \rrbracket^\perp, w' : \llbracket C \rrbracket^\perp \parr \llbracket D \rrbracket}$$

Figure 3.D.2: CEGV, functional fragment, Part two

$$\left[\!\!\left[\begin{array}{c}\textsc{Send}\\ \dfrac{\Delta \vdash Q : C \qquad \Gamma \vdash P : !C.D}{\Gamma, \Delta \vdash \mathsf{send}\ Q\ P : D}\end{array}\right]\!\!\right] :=$$

$$\dfrac{\llbracket P \rrbracket_z \vdash \llbracket \Gamma \rrbracket^\perp, z : \llbracket C \rrbracket^\perp \parr \llbracket D \rrbracket \qquad \dfrac{\llbracket Q \rrbracket_c \vdash \llbracket \Delta \rrbracket^\perp, c : \llbracket C \rrbracket \qquad d \leftrightarrow d' \vdash d : \llbracket D \rrbracket, d' : \llbracket D \rrbracket^\perp}{d'[c].\, (\llbracket Q \rrbracket_c \mid d \leftrightarrow d') \vdash \llbracket \Delta \rrbracket^\perp, d : \llbracket D \rrbracket, d' : \llbracket C \rrbracket \otimes \llbracket D \rrbracket^\perp}}{\nu z d'\ (\llbracket P \rrbracket_z \mid d'[c].\, (\llbracket Q \rrbracket_c \mid d \leftrightarrow d')) \vdash \llbracket \Gamma \rrbracket^\perp, \llbracket \Delta \rrbracket^\perp, d : \llbracket D \rrbracket}$$

$$\left[\!\!\left[\begin{array}{c}\textsc{Recv}\\ \dfrac{\Gamma \vdash P : ?C.D}{\Gamma \vdash \mathsf{recv}\ P : C \otimes D}\end{array}\right]\!\!\right]_z := \llbracket P \rrbracket_z \vdash \llbracket \Gamma \rrbracket^\perp, z : \llbracket C \rrbracket \otimes \llbracket D \rrbracket$$

$$\left[\!\!\left[\begin{array}{c}\textsc{Term}\\ \dfrac{\Gamma \vdash P : \mathsf{end}_?}{\Gamma \vdash \mathsf{terminate}\ P : \mathbf{1}}\end{array}\right]\!\!\right]_z := \llbracket P \rrbracket_z \vdash \llbracket \Gamma \rrbracket^\perp, z : \mathbf{1}$$

$$\left[\!\!\left[\begin{array}{c}\textsc{Pop}\\ \dfrac{\Gamma, x : C_S \vdash P : \mathsf{end}_!}{\Gamma \vdash \mathsf{pop}_x(P) : \overline{C_S}}\end{array}\right]\!\!\right]_z := \dfrac{\llbracket P \rrbracket_z \vdash \llbracket \Gamma \rrbracket^\perp, x : \llbracket C \rrbracket_S^\perp, z : \perp}{\nu z z'\ (\llbracket P \rrbracket_z \mid \mathbf{0}_{z'}) \vdash \llbracket \Gamma \rrbracket^\perp, x : \llbracket C \rrbracket_S^\perp}$$

$$\left[\!\!\left[\begin{array}{c}\textsc{Push}\\ \dfrac{\Gamma \vdash P : C_S}{\Gamma, x : \overline{C_S} \vdash \mathsf{push}_x(P) : \mathsf{end}_!}\end{array}\right]\!\!\right]_z := \dfrac{\llbracket P \rrbracket_x \vdash \llbracket \Gamma \rrbracket^\perp, x : \llbracket C_S \rrbracket}{z().\, \llbracket P \rrbracket_x \vdash \llbracket \Gamma \rrbracket^\perp, x : \llbracket C \rrbracket_S^\perp, z : \perp}$$

Figure 3.D.3: CEGV, typing rules of sessions

$$\left[\!\!\left[\begin{array}{c}\textsc{LetE}\\ \dfrac{\Gamma \vdash P : D \qquad \Delta, d : D \mid \Theta \vdash Q : E}{\Gamma, \Delta \mid \Theta \vdash \mathsf{let}\ d\ =\ P\ \mathsf{in}\ \ Q : E}\end{array}\right]\!\!\right]_{xzC} :=$$

$$\dfrac{\llbracket P\rrbracket_d \vdash \llbracket \Gamma\rrbracket^{\perp}, d : \llbracket D\rrbracket \qquad \llbracket Q\rrbracket_{xzC} \vdash \llbracket \Delta\rrbracket^{\perp}, d' : \llbracket D\rrbracket^{\perp}, x : \boxdot^{\llbracket\Theta\rrbracket} C, z : \llbracket E\rrbracket \otimes \diamonddot^{\llbracket\Theta\rrbracket^{\perp}} C^{\perp}}{\nu dd' \left(\llbracket P\rrbracket_d \mid \llbracket Q\rrbracket_{xzC}\right) \vdash \llbracket\Gamma\rrbracket^{\perp}, \llbracket\Delta\rrbracket^{\perp}, x : \boxdot^{\llbracket\Theta\rrbracket} C, z : \llbracket E\rrbracket \otimes \diamonddot^{\llbracket\Theta\rrbracket^{\perp}} C^{\perp}}$$

$$\left[\!\!\left[\begin{array}{c}\textsc{DoUntil}\\ \dfrac{e : E \mid \Theta \vdash Q : E + D}{e : E \mid \Theta \vdash \mathsf{doUntil}\ e.Q : D}\end{array}\right]\!\!\right]_{xz'C} :=$$

$$\dfrac{\dfrac{\llbracket Q\rrbracket_{xzD} \vdash e : \llbracket E\rrbracket^{\perp}, x : \boxdot^{\llbracket\Theta\rrbracket} C, z : (\llbracket E\rrbracket \oplus \llbracket D\rrbracket) \otimes \diamonddot^{\llbracket\Theta\rrbracket^{\perp}} C^{\perp} \qquad x \leftrightarrow z' \vdash x : \boxdot^{\llbracket\Theta\rrbracket} C, z' : \diamonddot^{\llbracket\Theta\rrbracket^{\perp}} C^{\perp}}{P := z'\langle c\rangle.\, x \leftrightarrow z' \vdash c : \llbracket D\rrbracket^{\perp}, x : \boxdot^{\llbracket\Theta\rrbracket} C, z' : \llbracket D\rrbracket \otimes \diamonddot^{\llbracket\Theta\rrbracket^{\perp}} C^{\perp}}}{\boxdot x(e)\{z.\, \llbracket Q\rrbracket_{xzD}\}c.\, P \vdash e : \llbracket E\rrbracket^{\perp}, x : \boxdot^{\llbracket\Theta\rrbracket} C, z' : \llbracket D\rrbracket \otimes \diamonddot^{\llbracket\Theta\rrbracket^{\perp}} C^{\perp}}\ \textsc{BoR}$$

$$\textsc{RelaxE}\quad \dfrac{\Gamma \mid \Theta \vdash P : D}{\Gamma \mid \Theta, \mathsf{i} : (A : B) \vdash P := \mathsf{open}(\mathsf{relax}(\mathsf{close}(P))) : D}\ \text{PROPOSITION 30}$$

$$\textsc{Close}\quad \dfrac{\dfrac{\Gamma \mid \Theta \vdash P : D}{\Gamma \vdash \mathsf{inst}(P) : (D \multimap \boxdot^{\Theta} D) \multimap \boxdot^{\Theta} D} \qquad \vdash \mathsf{unit} : D \multimap \boxdot^{\Theta} D}{\Gamma \vdash \mathsf{close}(P) := \mathsf{inst}(P)(\mathsf{unit}) : \boxdot^{\Theta} D}$$

Open is derived by cutting x' of $\llbracket P\rrbracket_{x'}$ with y' of the following, where $S := \diamonddot^{\Theta^{\perp}} C^{\perp}$

$$\dfrac{\dfrac{R := y(z).\, z\langle y\rangle.\, f \leftrightarrow z \vdash f : S^{\perp}, y : (D \otimes S) \parr D^{\perp}}{\diamonddot y\{i.\, i \leftrightarrow i', za'z'.\, \mathsf{absorb}_{a'z'z}, f.\, R\} \vdash y : \diamonddot^{\Theta^{\perp}}((D \otimes S) \parr D^{\perp}), i' : S^{\perp}} \qquad \mathsf{lift}_{xzy'} \vdash x : (\diamonddot^{\Theta^{\perp}}((D \otimes S) \parr D^{\perp}))^{\perp}, y' : \diamonddot^{\Theta^{\perp}}(D^{\perp}), z : D \otimes S}{\nu xy \cdots \vdash y' : \diamonddot^{\Theta^{\perp}} D^{\perp}, z : D \otimes S, i' : S^{\perp}}$$

Figure 3.D.4: CEGV, omitted typing rules of effects

Proposition 37. *The operation $\overline{}$ is an involution; that is $\overline{\overline{C_S}} = C_S$.*

Proof. By observing the definition of duality in CEGV. □

The following implication states that an emitter that does not use any effect can be converted into a pure value. They are not used in our paper but still interesting.

Proposition 38 (Purify)**.**

$$\boxdot^\bullet C \multimap C \qquad\qquad C \multimap \diamonddot^\bullet C$$

Proof. We will derive them in CEGV instead of CELL for simplicity. The first can be derived by HANDLER and the second by RUNNER (which we omit)

$$\mathsf{purify} := \;\vdash \mathsf{handler}\{i.i, a'z'.\_, y.y\} : \boxdot^\bullet C \multimap C$$

□

Note that the other direction can be given by the UNIT rule: $\vdash \mathsf{unit} : C \multimap \boxdot^\bullet C$. They together form an equivalence, as stated by the next theorem.

Proposition 39.

- *For any $\Gamma \vdash P : C$, we have $\mathsf{purify}(\mathsf{unit}(P)) \approx P$*
- *For any $\Gamma \vdash P : \boxdot^\bullet C$, we have $\mathsf{unit}(\mathsf{purify}(P)) \approx P$*

Proof. We first translates unit and purify to CELL (unfolding $\parr$):

$$\mathsf{unit'} := \boxdot x[].\, x \leftrightarrow c' \vdash c' : C^\perp, x : \boxdot^\bullet C$$
$$\mathsf{purify'} := \diamonddot y\{i.\, i \leftrightarrow c'', zaz'.\_, f.\, y \leftrightarrow f\} \vdash y : \diamonddot^\bullet C^\perp, c'' : C$$

Now for the first bisimilarity, we need to prove that for any $P' \vdash \Gamma^\perp, c : C$, we have

$$\nu cc'\, \nu xy\, (P' \mid \mathsf{unit'} \mid \mathsf{purify'}) \approx P'[c''/c]$$

which follows by lemma 27 and the fact that LHS would τ-transition to RHS.

For the second bisimilarity, we need to prove that for any $P' \vdash \Gamma^\perp, x' : \boxdot^\bullet C$, we have

$$\nu c'c''\, \nu x'y\, (P' \mid \mathsf{unit'} \mid \mathsf{purify'}) \approx P'[x/x']$$

Let $R :=\approx \cup \{(\nu c'c''\, \nu x'y\, (P' \mid \mathsf{unit'} \mid \mathsf{purify'}), P'[x/x']) \mid P' \vdash \Gamma, x' : \boxdot^\bullet C\}$. We will prove R to be bisimulation and thus $R \subset \approx$.

- For labels l s.t. $\textsc{Fn}(l) \in \Gamma$, then for LHS it must be by RES and say $P' \xrightarrow{l} Q$. That means LHS transitions to

  $$\nu c'c''\, \nu x'y\, (Q \mid \mathsf{unit'} \mid \mathsf{purify'})$$

  and RHS transitions to $Q[x/x']$. Note that LHS and RHS after transition are still in R.

- For label $\boxdot x[]$, say RHS transitions to $Q[x/x']$, and LHS transitions to
$$Q' := \nu c'c'' \, \nu x'y \, (P' \mid x \leftrightarrow c' \mid \mathsf{purify'})$$
, which further τ-transitions to $Q[x/x']$. As a result, $(Q', Q[x/x']) \in R$ by lemma 27

□

Proof of proposition 28. We omit the types for simplicity

- (left identity) LHS translates to
$$\nu z_0 x_1 \, (z_0[d'].\, (\llbracket P \rrbracket_{d'} \mid x_0 \leftrightarrow z_0) \mid x_1(d).\, \llbracket Q \rrbracket_{x_1 z_1 C})$$
RHS translates to
$$\nu dd' \, (\llbracket P \rrbracket_{d'} \mid \llbracket Q \rrbracket_{x_0 z_1 C})$$
We easily have LHS τ-transition to RHS, and thus follows by lemma 27.

- (right identity) LHS translates to
$$\nu z x' \, (\llbracket P \rrbracket_{xzC} \mid x'(d).\, z'[d'].\, (d \leftrightarrow d' \mid x' \leftrightarrow z'))$$
RHS translates to
$$\llbracket P \rrbracket_{xz'C}$$
Consider how $\llbracket P \rrbracket_{xz'C}$ will transition. First case is $\xrightarrow{z'[c]} Q$, then we should have $\llbracket P \rrbracket_{xzC} \xrightarrow{z[c]} Q[z/z']$, and therefore LTS would τ-transition into
$$\nu z x' \, \nu cd \, (Q[z/z'] \mid z'[d'].\, (d \leftrightarrow d' \mid x' \leftrightarrow z'))$$
which then $\xRightarrow{z'[d'].}$ into $Q[d'/c]$. On the other hand, RHS simply $\xrightarrow{z[c]}$ Q. Therefore the two are bisimilar.
For any other cases, LHS will simulate RHS by Res.

- (commutativity) RHS translates to
$$\nu z_0 x_1 \, (\llbracket P \rrbracket_{x_0 z_0 C} \mid x_1(d).\, \nu z_1 x_2 \, (\llbracket Q \rrbracket_{x_1 z_1 C} \mid x_2(e).\, \llbracket R \rrbracket_{x_2 z_2 C}))$$
RHS translates to
$$\nu z_1 x_2 \, (\nu z_0 x_1 \, (\llbracket P \rrbracket_{x_0 z_0 C} \mid x_1(d).\, \llbracket Q \rrbracket_{x_1 z_1 C}) \mid x_2(e).\, \llbracket R \rrbracket_{x_2 z_2 C})$$
which are trivially bisimilar.

□

We conjecture the following properties about Spawn, but did not prove them due to the computational complexity.

- A spawned process and a continuation thereof are on an equal footing. That is for any $\Gamma \mid \Theta \vdash P : \mathbf{1}$ and $\Delta \mid \Theta \vdash Q : \mathbf{1}$ we have

$$\Gamma, \Delta \mid \Theta \vdash \mathsf{spawn}(\mathsf{close}(P)); Q \approx \mathsf{spawn}(\mathsf{close}(Q));\ P : \mathbf{1}$$

- Spawning a trivial emitter does nothing:

$$\Gamma \mid \Theta \vdash \mathsf{spawn}(\mathsf{unit}(\star)) \approx \mathsf{return}(\star) : \mathbf{1}$$

We can portray the behaviours of example 8 using bisimilarities, as follows.

Theorem 40 (One-sided Equality Theory)**.** *We have the following facts in CEGV.*

- *Given* $\Gamma \vdash P : \mathbf{1} + X$ *we have* $[\![\mathsf{cell}^X(P)]\!]_y \xrightarrow{\Diamonddot y\text{–}(a')} \xrightarrow{a'(\mathrm{L})} \approx [\![(P, \mathsf{cell}^X(\mathsf{inl}\,\star))]\!]_y$*, which says getting from a cell containing* P *(which might be empty) returns* P *and leaves the cell empty.*

- $[\![\mathsf{cell}^X(\mathsf{inl}\,\star)]\!]_y \xrightarrow{\Diamonddot y\text{–}(a')} \xrightarrow{a'(\mathrm{R})} \approx [\![(\mathsf{inl}\,\star, \mathsf{cell}^X(\mathsf{inr}\,a'))]\!]_y$ *which says putting to an empty cell will fill the cell and returns empty.*

- *Given* $\Gamma \vdash P' : X$ *we have*

$$[\![\mathsf{cell}^X(\mathsf{inr}\,P')]\!]_y \xrightarrow{\Diamonddot y\text{–}(a')} \xrightarrow{a'(\mathrm{R})} \approx$$
$$[\![(\mathsf{inr}\,a', \mathsf{cell}^X(\mathsf{inr}\,P'))]\!]_y$$

which says putting P *to a non-empty cell will not change the cell and returns* P

Proof. We first translate the statements into CELL:

- Given $P \vdash \Gamma, i : S_X$, we have

$$cell_y^X(i.P) \xrightarrow{\Diamonddot y\text{–}(a')} \xrightarrow{a'[\mathrm{L}]} \approx$$
$$a'().\,y[i].\,(P \mid cell_y^X) \vdash \Gamma, a' : \bot, y : S_X \otimes \Diamonddot\Theta^{\perp}{}_X S_X$$

- we have

$$cell_y^X \xrightarrow{\Diamonddot y\text{–}(a')} \xrightarrow{a'[\mathrm{R}]} \approx$$
$$y[i].\,(i[\mathrm{L}].\,\mathbf{0}_i \mid cell_y^X(i.i[\mathrm{R}].\,a' \leftrightarrow i)) \vdash \Gamma, a' : X^{\perp}, y : S_X \otimes \Diamonddot\Theta^{\perp}{}_X S_X$$

- Given $P' \vdash \Gamma, i : X$, we have

$$cell_y^X(i.i[\mathrm{R}].\,P') \xrightarrow{\diamond y\text{–}(a')}\xrightarrow{a'[\mathrm{R}]}\approx$$
$$y[b].\,(b[\mathrm{R}].\,a' \leftrightarrow b \mid cell_y^X(i.i[\mathrm{R}].\,P')) \vdash \Gamma, a' : X^\perp, y : S_X \otimes \diamond\Theta^\perp{}_X S_X$$

And then we prove them one by one:

- We have LHS

$$cell_y^X(i.P) \xrightarrow{\diamond y\text{–}(a')}\xrightarrow{a'[\mathrm{L}]} \nu i'z'\,\nu iz\,(Q_{get} \mid \mathcal{D} \mid P)$$

  whose only possible further transition (by tedious computation) is (implicitly using some alpha-renaming)

$$\xrightarrow{a'()}\overset{\tau}{\Longrightarrow}\xrightarrow{y[i]} \nu i'z'\,(P \mid z'[\mathrm{L}].\,\mathbf{0}_{z'} \mid cell_y^X(i.i \leftrightarrow i'))$$

  note that the only possible transition of RHS $a().\,y[i].\,P \mid cell_y^X$ is

$$\xrightarrow{a()}\xrightarrow{y[i]} P \mid cell_y^X$$

  We therefore only need to further prove

$$\nu i'z'\,(z'[\mathrm{L}].\,\mathbf{0}_{z'} \mid cell_y^X(i.i \leftrightarrow i')) \approx cell_y^X(i.i[\mathrm{L}].\,\mathbf{0}_i)$$

  which is easy, because on both sides, the only possible transition is $\diamond y\text{–}(a')$, which on LHS gives (using some alpha-renaming)

$$\nu i'z'\,(z'[\mathrm{L}].\,\mathbf{0}_{z'} \mid \nu iz\,\nu i''z''\,Q[z''/z'] \mid \mathcal{D}[i''/i'] \mid i \leftrightarrow i')$$
$$\xrightarrow{\tau}\nu iz\,\nu i''z''\,(Q[z''/z'] \mid D[i''/i'] \mid i[\mathrm{L}].\,\mathbf{0}_i)$$

  which is same as RHS after the $\diamond y\text{–}(a')$ transition. Citing lemma 27 and we are finished.

- We have LHS

$$cell_y^X \xrightarrow{\diamond y\text{–}(a')}\xrightarrow{a'[\mathrm{R}]} \nu i'z'\,\nu iz\,(Q_{put} \mid \mathcal{D} \mid i[\mathrm{L}].\,i[].\,\mathbf{0})$$

  which further transitions

$$\overset{\tau}{\Longrightarrow}\xrightarrow{y[i]}\overset{\tau}{\Longrightarrow} \nu i'z'\,(cell_y^X(i.i \leftrightarrow i') \mid i[\mathrm{L}].\,\mathbf{0}_i \mid z'[\mathrm{R}].\,a' \leftrightarrow z')$$

  while the RHS transitions

$$\xrightarrow{y[i]} (i[\mathrm{L}].\,\mathbf{0}_i \mid cell_y^X(i.i[\mathrm{R}].\,a' \leftrightarrow i))$$

thus only to show is

$$\nu i' z' \, (cell_y^X(i.i \leftrightarrow i') \mid z'[\mathrm{R}].\, a' \leftrightarrow z') \approx cell_y^X(i.i[\mathrm{R}].\, a' \leftrightarrow i)$$

which is easy, because on both sides, the only possible transition is $\diamond y\text{–}(a')$, which on LHS gives

$$\begin{aligned} &\nu i' z' \, (\nu iz \, \nu i'' z'' \, (Q[z''/z'] \mid \mathcal{D}[i''/i'] \mid i \leftrightarrow i') \mid z'[\mathrm{R}].\, a' \leftrightarrow z') \\ \xrightarrow{\tau} &\nu iz \, \nu i'' z'' \, (Q[z''/z'] \mid \mathcal{D}[i''/i'] \mid i[\mathrm{R}].\, a' \leftrightarrow i) \end{aligned}$$

which is exactly as RHS after $\diamond y\text{–}(a')$ transition. Citing lemma 27 and we are done.

- We have LHS

$$cell_y^X(i.i[\mathrm{R}].\, P') \xrightarrow{\diamond y\text{–}(a')} \xrightarrow{a'[\mathrm{R}]} \nu i' z' \, \nu iz \, (Q_{put} \mid \mathcal{D} \mid i[\mathrm{R}].\, P')$$

which further transitions

$$\overset{\tau}{\Longrightarrow} \xrightarrow{y[b]} \overset{\tau}{\Longrightarrow} \nu i' z' \, \nu iz \, (P' \mid z'[\mathrm{R}].\, z \leftrightarrow z' \mid b[\mathrm{R}].\, a' \leftrightarrow b \mid cell_y^X(i.i \leftrightarrow i'))$$

while the RHS transitions

$$\xrightarrow{y[b]} (b[\mathrm{R}].\, a' \leftrightarrow b \mid cell_y^X(i.i[\mathrm{R}].\, P'))$$

thus only to show is

$$\nu i' z' \, \nu iz \, (P' \mid z'[\mathrm{R}].\, z \leftrightarrow z' \mid cell_y^X(i.i \leftrightarrow i')) \approx cell_y^X(i.i[\mathrm{R}].\, P')$$

which is easy, because the only possible transition on both sides is $\diamond y\text{–}(a')$, which on LHS gives

$$\begin{aligned} &\nu i' z' \, \nu iz \, (P' \mid z'[\mathrm{R}].\, z \leftrightarrow z' \mid \nu i'' z'' \, \nu i''' z''' \, (\mathcal{D}[i'''/i'] \mid i'' \leftrightarrow i' \mid Q[z'''/z'][z''/z])) \\ \xrightarrow{\tau} &\nu iz \, (P' \mid \nu i'' z'' \, \nu i''' z''' \, (\mathcal{D}[i'''/i'] \mid i''[\mathrm{R}].\, z \leftrightarrow i'' \mid Q[z'''/z'][z''/z])) \\ \approx &\nu i'' z'' \, \nu i''' z''' \, (\mathcal{D}[i'''/i'] \mid \nu iz \, (P' \mid i''[\mathrm{R}].\, z \leftrightarrow i'') \mid Q[z'''/z'][z''/z]) \end{aligned}$$

Compared to RHS after $\diamond y\text{–}(a')$, we only need to show (after some alpha-renaming)

$$i''[\mathrm{R}].\, P'[i''/i] \approx \nu iz \, (P' \mid i''[\mathrm{R}].\, z \leftrightarrow i'')$$

which is trivial because the only possible transition on both sides is $i''[\mathrm{R}]$, after which LHS gives $P'[i''/i]$, and RHS (via some τ transition) also gives $P'[i''/i]$.

$\square$

Example 13 (Linear Exception)**.** Exception is a common example of typical effect systems. This is however difficult in our system, because we are not allowed to discard continuation which is central for exception handling. As a result, we must use the continuation, which brings the second issue that we are not able to provide absurdity to continuation. Recall that typically continuation receives absuridty which can be casted to any type in need; this is an important feature of exception. Absurdity in linear logic corresponds to the additive units, which are always omitted from the CP intepretation of linear logic; To amend this would worth a standalone paper and we leave it to future work. Here we take the easy way where we pass $B := \mathbf{1}$ to the continuation, and simply discard the returning value of the continuation.

$$S := E + X \qquad A := E \qquad B := \mathbf{1} \qquad D := X \qquad \Theta := \mathsf{raise} : (A : B)$$

To handle exception at a', we resume the continuation with $\star$ and disgards the return value; instead we will use a' as the return value. Note that we require $E+X$ to be a unlimited type (positive type) for silently dropping; this is always the case in typical functional programming.

$$\begin{aligned} &a' : A, z' : B \multimap S \vdash Q : S \\ &Q := \mathsf{let}\ \_ = z' \star \mathsf{in} && \text{(silently drops the return value)} \\ &\quad \mathsf{inl}\ a' && \text{(record error in result)} \end{aligned}$$

And we have

$$\vdash \mathsf{catch} := \mathsf{handler}\{i.i, a'z'.Q, y.\mathsf{inr}\ y\} : y : \boxdot^{\Theta} D \multimap S$$

On the other hand, we define main program (assuming $\vdash e_0 : E$ and $\vdash x_0 : X$) which raises exception before returning.

$$\begin{aligned} &\mid \Theta \vdash M : D \\ &M := \mathsf{emit}^{\mathsf{raise}}(e_0); && \text{(discard the unit given by the handler)} \\ &\quad \mathsf{return}(x_0) \end{aligned}$$

If we run M inside catch:

$$\vdash \mathsf{catch}(\mathsf{close}(M)) \approx \mathsf{inl}\ e_0 : S$$

we get some S which will be bisimilar to $\mathsf{inl}\ e_0$; the bisimilarity can be easily proved by lemma 27.

While we can somewhat express exception, non-determinism (which calls continuation multiple times with different response) would be certainly not possible. Linearity of continuation is discussed in section 3.8.

Example 14 (Effects as a Service (EaaS))**.** In example 10, instead of incrementing a number, we can allow client to perform effects directly by setting $X_S := \diamonddot\Theta\mathsf{end}_?$ and make serve simply forward all Θ-effects from the client to the server.

$$x : X_S \mid \Theta \vdash \mathsf{serve} := x \leftarrow \mathsf{open}(x);\ \mathsf{return}(\mathsf{terminate}\ x) : \mathbf{1}$$

Note in this case, cell^{X_S} stores $\boxdot^{\Theta}\mathsf{end}_?$; i.e. we have effects whose request/response are effectful computations. This is only possible because the latter are merely types.

Note that example 10 still lacks a program that put new sessions to cell. Depending on the scenario we want to model, there are two approaches. If we assume the client process to be some other emitter sharing the cell with the server, it can simply put sessions itself, which is simple and omitted. If we assume the client process to be isolated from the server and only talks to the server via a channel, one tends to use coexponential [Qian et al., 2021], but for simplicity we use list to model client queue. We could assume list as primitive, but we will define it for self-containment.

Example 15 (Encoding List)**.** We can think $[X]$ a list of X as an emitter that for many times emits X while expecting only trivial response. Define

$$\Theta := \_ : (X : \mathbf{1}) \qquad\qquad [X] := \diamonddot^{\Theta}\mathbf{1}$$

We now have:

$$\vdash [] := \mathit{unit}(\star) : [X] \qquad \frac{\Gamma \vdash P : X \qquad \Delta \vdash Q : [X]}{\Gamma, \Delta \vdash P :: Q := \mathsf{inst}(\mathsf{emit}{-}(P))(\lambda\_.\,Q) : [X]}$$

$$\frac{\Gamma \vdash P : S \qquad z : S, a' : X \vdash Q : S}{\Gamma \vdash \mathsf{fold}(P, za'.Q) := \mathsf{handler}\{i.P, a'z'.\mathsf{let}\ z\ =\ z'(\star)\ \mathsf{in}\ \ Q, y.y\} : [X] \multimap S}$$

Logically $[X] \equiv \text{¿}X$, but semantically the latter allows clients permutation while the former does not, which is essential for racy client acception. This is however not a problem for modeling most client/server, because while the clients acception is deterministic, their interaction with the server emits effects on the server and races with each other.

We are now at a position to define the process that accepts client sessions X_S and put them to the cell^{X_S}.

Example 16 (Accepting client connections)**.** We want to fold the list such that for each element (which is a connection X) we sput it to cell. Note that the list operations looks pure and do not mention effects at all, which is however not a problem. We use a similar trick as in example 7 and let $S := \boxdot^{\Omega}\mathbf{1}$ and we define

$$\begin{aligned} &\vdash P := \mathsf{unit}(\star) : S \\ z : S, a' : X &\vdash Q : S \\ Q :=&\, \mathsf{inst}(\mathsf{sput}_{a'})(\lambda\_.\, z) \\ &\vdash \mathsf{accepter} := \mathsf{fold}(P, za'.Q) : [X] \multimap S \end{aligned}$$

Note that in general the user of fold can pick any Ω they want, assuming the user can pick any S they want. Effect polymorphism[Brachthäuser et al., 2020b] is a heated area, and our approach of reducing it to type polymorphism has limitations; for example, the function $map : (X \multimap Y) \multimap [X] \multimap [Y]$ cannot support effects by simply instantiating X, Y to appropriate types. We leave it to future works.

We now define (Recall that Θ represents database IO):

$$\begin{aligned} \mathsf{server} :=&\, \mathsf{using\ cell}^{X_S}(\mathsf{inl}\ \star)(\\ &\quad \mathsf{spawn}(\mathsf{accepter}(y)); \\ &\quad \mathsf{workers}) \\ \Gamma, y : [X_S] \mid \Theta &\vdash \mathsf{server} : \mathbf{1} \end{aligned}$$

Example 17 (Bidirectional Effects)**.** Zhang et al. [2020] introduced bidirectional effects which means handler might emits effects themselves. In contrast to example 7 where the effects are handled by more primitive co-emitter, here the effect are handled by the effectful computation that emit effect in the first place. Our system can easily express this by setting the effect response to be an emitter. Note this is only possible because the latter is merely a type, which allows higher-order effects.

We consider a simple example where a main program would try to get some value X. The effect will be handled and in fact cause some exception E back to the main program who will handle the exception. We define

$$\Omega := \mathsf{get} : (\mathbf{1} : \boxdot^{\Theta} X) \qquad\qquad \Theta := \mathsf{raise} : (E : \mathbf{1})$$

We first define the main program:

$$\begin{aligned} &\mid \Omega \vdash \mathsf{main} : E + X \\ \mathsf{main} :=&\, x \leftarrow \mathsf{emit}^{\mathsf{get}}(\star); \\ &\quad \mathsf{return}(\mathsf{catch}(x)) \end{aligned}$$

where catch is defined in example 13. We then define the Ω-runner:

$$\begin{aligned} &\vdash P := \star : S \\ z : S, a' : \mathbf{1} &\vdash Q_{get} := (\mathsf{close}(\mathsf{emit}^{\mathsf{raise}}(e_0);\ \mathsf{return}(x_0)), \star) : \boxdot^{\Theta} X \otimes S \\ f : S &\vdash R := \star : \mathbf{1} \\ &\vdash \mathsf{runner} := \mathsf{runner}\{P, za'.Q, f.R\} : \diamondtimes^{\Omega}\mathbf{1} \end{aligned}$$

And we connect runner with main:

$$\begin{aligned}&\vdash \mathsf{let}\ (s, y) \ = \ \mathsf{coinst}(\mathsf{main})(\mathsf{runner})\ \mathsf{in}\\ &\ \ \mathsf{let}\ \_ \ = \ \mathsf{counit}(y)\ \mathsf{in}\ s\\ &:E + X\end{aligned}$$

we further claim the above is bisimilar to $\mathsf{inl}\ e_0$.

Example 18 (Positive Cell)**.** A more traditional cell would (in contrast to linear cell through out the paper) allow overwriting (destruction) and repeated reading (duplication), in which case the content has to be an unlimited type X_U. This was the case in Rocha and Caires [2021] and can be reproduced in our system. We choose to implement it in CEGV, as unlimited types are more natural in CEGV than in CELL. Assuming initial value $\Gamma \vdash P : X_U$ supplied by the user. We first specify the effects:

$$\Theta := \mathsf{read} : (\mathbf{1} : X_U), \mathsf{write} : (X_U : \mathbf{1})$$

We then define the positive cell using Runner with internal state $S := X_U$. For reading, we duplicate the old state z to two copies, one used as new state, one given to user. For writing, we first discard the old state, and forward the user input to the new state. However, both duplication and discarding are implicit thanks to Weaken and Contract. In finalization we simply return the internal state.

$$\begin{aligned}z : S, a' : \mathbf{1} &\vdash Q^{\mathsf{read}} := (z, z) : X_U \otimes S\\ z : S, a' : X_U &\vdash Q^{\mathsf{write}} := (\star, a') : \mathbf{1} \otimes S\end{aligned}$$

Combining everything:

$$\mathsf{pcell}^{X_U}(P) := \mathsf{runner}\{P, za'.Q, f.f\} \vdash \Gamma, y : \diamond^{\Theta} S$$

This example is less interesting than linear cell as it cannot store non-positive types such as sessions.

Example 19 (Dining Philosophers)**.** Dining philosophers has been difficult to express in linear logic due to its cyclic nature. We will represent a chopstick as a linear cell of $\mathbf{1}$. Non-empty cell means the chopstick is occupied. To allow each philosopher to access multiple $\mathsf{cell}^{\mathbf{1}}$, we follow example 11 and define the following prefixed version of sput and sget

$$\begin{aligned}\mid \mathsf{i.get} : (\mathbf{1} : \mathbf{1} + X) &\vdash \mathsf{sget}^{\mathsf{i}} := \mathsf{let}\ e \ = \ \star\ \mathsf{in}\ \ \mathsf{doUntil}\ e.\mathsf{emit}^{\mathsf{i.get}}(\star) : X\\ e : X \mid \mathsf{i.put} : (X : X + \mathbf{1}) &\vdash \mathsf{sput}^{\mathsf{i}}_{e} := \mathsf{doUntil}\ e.\mathsf{emit}^{\mathsf{i.put}}(e) : \mathbf{1}\end{aligned}$$

We first define a single philosopher; they would acquire the left and right sticks in order, and then release them.

$$\begin{aligned}\Omega \ &:= \ \mathsf{i.get} : (\mathbf{1} : \mathbf{1} + \mathbf{1}), \mathsf{i.put} : (\mathbf{1} : \mathbf{1} + \mathbf{1}), \mathsf{j.get} : (\mathbf{1} : \mathbf{1} + \mathbf{1}), \mathsf{j.put} : (\mathbf{1} : \mathbf{1} + \mathbf{1})\\ \mid \Omega &\vdash \mathsf{nerd}^{ij} := \mathsf{let}\ u \ = \ \star\ \mathsf{in}\ \mathsf{sput}^{\mathsf{i}}_{u}; \mathsf{sput}^{\mathsf{j}}_{u}; \mathsf{sget}^{\mathsf{i}}; \mathsf{sget}^{\mathsf{j}}; \mathsf{return}(\star) : \mathbf{1}\end{aligned}$$

We first derive the following syntax sugar based on example 11 and USING that silently discards the coemitter after using it, given that it returns an unlimited value. This is to simplify the main program.

$$\frac{\Gamma \mid \Omega \vdash P : \diamondsuit^{\Theta} E_U \qquad \Delta \mid \mathsf{j}.\Theta, \Omega \vdash Q : D}{\Gamma, \Delta \mid \Omega \vdash \mathsf{using*}\ \mathsf{j}.P(Q) := (d, y) \leftarrow \mathsf{using}\ \mathsf{j}.P(Q); \mathsf{let}\ \_ = \mathsf{coinst}(y)\ \mathsf{in}\ \mathsf{return}(d) : D}$$

We now write the main program; we start by allocating two cells prefixed with i and j with empty initial state. We then spawn two philosophers with different order of cells.

$$\begin{aligned}
&\mid\ \vdash M : \mathbf{1} \\
M :=&\mathsf{using*}\ \mathsf{i.cell}^{\mathbf{1}}(\mathsf{inl}\ \star)(&& \text{(allocate empty cell and use it)} \\
&\mathsf{using*}\ \mathsf{j.cell}^{\mathbf{1}}(\mathsf{inl}\ \star)(&& \text{(allocate empty cell and use it)} \\
&\mathsf{spawn}(\mathsf{nerd}^{ij}); \mathsf{spawn}(\mathsf{nerd}^{ji})))
\end{aligned}$$

Here using* silently destroys (instead of returning) the cells after using them. Note that M is an effectful computation with empty effect environment.

Remark 3 (Cell is not Reference). example 9 gives the impression that our Cell^X is similar to references [Leroy et al., 2022]. However, Cell^X being a coemitter is not a positive type, and thus cannot be refered to more or less than once. This is why the above nerd is parameterized over i and j instead of taking two $\mathsf{Cell}^{\mathbf{1}}$ as arguments, since each $\mathsf{Cell}^{\mathbf{1}}$ would be refered to by both nerd.

Bibliography

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987a. ISSN 03043975. doi: 10.1016/0304-3975(87)90045-4.

Jean-Yves Girard. Linear logic and parallelism. In *Mathematical Models for the Semantics of Parallelism*, pages 166–182. Springer, 1987b.

Jean-Yves Girard. Towards a geometry of interaction. *Contemporary Mathematics*, 92(69-108):6, 1989.

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

G. Bellin and P. J. Scott. On the π-calculus and linear logic. *Theoretical Computer Science*, 135(1):11–65, 1994. ISSN 03043975. doi: 10.1016/ 0304-3975(94)00104-9.

Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and computation*, 100(1):1–40, 1992. doi: 10. 1016/0890-5401(92)90008-4.

Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory*, CONCUR'10, page 222–236, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3642153747. doi: 10.5555/1887654.1887670.

Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, Berlin, Heidelberg, 1993. doi: 10.1007/3-540-57208-2_35.

Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. 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)*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, Berlin, Heidelberg, 1998. doi: 10.1007/BFb0053567.

Davide Sangiorgi. π-calculus, internal mobility, and agent-passing calculi. *Theoretical Computer Science*, 167(1-2):235–274, 1996.

Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theoretical computer science*, 195(2):205–226, 1998.

Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, May 2014. ISSN 0956-7968. doi: 10.1017/S095679681400001X. ISBN: 1581137567.

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

Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. *Distributed Comput.*, 31(1):51–67, 2018. doi: 10.1007/s00446-017-0295-1. URL `https://doi.org/10.1007/s00446-017-0295-1`. Also: CONCUR 2014.

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

Fabrizio Montesi and Marco Peressotti. Classical transitions. 2018.

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

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

Fabrizio Montesi and Marco Peressotti. Linear logic, the π-calculus, and their metatheory: A recipe for proofs as processes, 2021. URL `https://arxiv.org/abs/2106.11818`.

Sam Lindley and J. Garrett Morris. Talking bananas: structural recursion for session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming - ICFP 2016*, pages 434–447, Nara, Japan, 2016. ACM Press. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951921.

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

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

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

Thomas Ehrhard and Farzad Jafarrahmani. Categorical models of linear logic with fixed points of formulas, 2021. To appear in the proceedings of LICS 2021.

Bernardo Toninho, Luís Caires, and Frank Pfenning. Corecursion and non-divergence in session-typed processes. In *International Symposium on Trustworthy Global Computing*, pages 159–175. Springer, 2014. doi: 10.1007/978-3-662-45917-1_11.

Farzaneh Derakhshan and Frank Pfenning. Circular proofs in first-order linear logic with least and greatest fixed points. 2020.

Robert Atkey, Sam Lindley, and J. Garrett Morris. Conflation Confers Concurrency. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World*, volume 9600 of *Lecture Notes in Computer Science*, pages 32–55. Springer International Publishing, 2016. ISBN 978-3-319-30935-4. doi: 10.1007/978-3-319-30936-1_2.

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

Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. *Acta Informatica*, 54:243–269, 2017. ISSN 0001-5903. doi: 10.1007/s00236-016-0285-y. Also: CONCUR 2015.

Luís Caires and Jorge A. Pérez. Linearity, control effects, and behavioral types. In Hongseok Yang, editor, *Programming Languages and Systems. ESOP 2017*, pages 229–259, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. ISBN 978-3-662-54434-1. doi: 10.1007/978-3-662-54434-1_9.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *Journal of the ACM*, 63(1):1–67, 2016. ISSN 0004-5411, 1557-735X. doi: 10.1145/2827695.

Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence Generalises Duality: A Logical Explanation of

Multiparty Session Types. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory (CONCUR 2016)*, volume 59 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 33:1–33:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-017-0. doi: 10.4230/LIPIcs.CONCUR.2016.33. URL `http://drops.dagstuhl.de/opus/volltexte/2016/6181`.

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

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

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

Naoki Kobayashi. Type Systems for Concurrent Programs. In Bernhard K. Aichernig and Tom Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 439–453, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi: 10.1007/978-3-540-40007-3_26. URL `http://www-kb.is.s.u-tokyo.ac.jp/~koba/papers/tutorial-type-extended.pdf`. Extended version.

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

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

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

Ornela Dardha and Simon J. Gay. A New Linear Logic for Deadlock-Free Session-Typed Processes. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, volume 10803 of *Lecture Notes in Computer Science*, pages 91–109, Cham, 2018. Springer International Publishing. doi: 10.1007/978-3-319-89366-2{_}5.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2008. ISBN 978-1-59593-689-9. doi: 10.1145/1328438.1328472.

Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016. ISSN 0960-1295, 1469-8072. doi: 10.1017/S0960129514000188.

Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic Multirole Session Types. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '11*, page 435. ACM Press, 2011. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926435.

Federico Aschieri and Francesco A. Genco. Par means parallel: Multiplicative linear logic proofs as concurrent functional programs. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi: 10.1145/3371086. URL `https://doi.org/10.1145/3371086`.

Andrea Tubella and Lutz Straßburger. Introduction to deep inference. 2019.

Lars Birkedal, Hans Bugge Grathwohl, Aleš Bizjak, and Ranald Clouston. The guarded lambda-calculus: Programming and reasoning with guarded recursion for coinductive types. *Logical Methods in Computer Science*, 12, 2017.

Zesen Qian, G. A. Kavvos, and Lars Birkedal. Client-server sessions in linear logic. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021. doi: 10.1145/3473567. URL `https://doi.org/10.1145/3473567`.

Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. distributed-systems.net, 3 edition, 2017. URL `https://www.distributed-systems.net/`.

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

Samson Abramsky, Simon J Gay, and Rajagopal Nagarajan. Interaction categories and the foundations of typed concurrent programming. In Manfred Broy, editor, *Deductive Program Design*, Nato ASI Subseries F, pages 35–113. Springer-Verlag Berlin Heidelberg, 1996. ISBN 3-540-60947-4. URL `http://www.springer.com/us/book/9783540609476`.

Samson Abramsky and Radha Jagadeesan. Games and Full Completeness for Multiplicative Linear Logic. *The Journal of Symbolic Logic*, 59(2):543, 1994. ISSN 00224812. doi: 10.2307/2275407.

Michael Barr. $*$-Autonomous categories and linear logic. *Mathematical Structures in Computer Science*, 1(2):159–178, 1991. ISSN 14698072. doi: 10.1017/S0960129500001274.

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

J. Y. Girard and Y. Lafont. Linear logic and lazy computation. In Hartmut Ehrig, Robert Kowalski, Giorgio Levi, and Ugo Montanari, editors, *TAPSOFT '87*, volume 250 of *Lecture Notes in Computer Science*, pages 52–66, Berlin/Heidelberg, 1987. Springer-Verlag. ISBN 978-3-540-17611-4. doi: 10.1007/BFb0014972.

Paul-André Melliès. Categorical Semantics of Linear Logic. In Pierre-Louis Curien, Hugo Herbelin, Jean-Louis Krivine, and Paul-André Melliès, editors, *Panoramas et synthèses 27: Interactive models of computation and program behaviour*. Société Mathématique de France, 2009. ISBN 978-2-85629-273-0. URL `http://www.pps.univ-paris-diderot.fr/~mellies/papers/panorama.pdf`.

Paul-André Melliès, Nicolas Tabareau, and Christine Tasson. An explicit formula for the free exponential modality of linear logic. *Mathematical Structures in Computer Science*, 28(7), 2018. ISSN 0960-1295, 1469-8072. doi: 10.1017/S0960129516000426. URL `https://www.cambridge.org/core/product/identifier/S0960129516000426/type/journal_article`.

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

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

Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, New York, NY, USA, 1999. ISBN 0-521-65869-1. doi: 10.5555/329902.

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

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

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, revised first edition, 2012. ISBN 978-0-12-397337-5. doi: 10.5555/2385452.

Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1-2):3–57, 1993b. ISSN 03043975. doi: 10.1016/0304-3975(93)90181-R. ISBN: 0304-3975.

Sam Lindley and J. Garrett Morris. Lightweight Functional Session Types. In Simon Gay and Antonio Ravara, editors, *Behavioural Types: from Theory to Tools*, River Publishers Series in Automation, Control and Robotics. River Publishers, 2017. doi: 10.13052/rp-9788793519817.

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

John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, Call-by-value, Call-by-need, and the Linear Lambda Calculus. *Electronic Notes in Theoretical Computer Science*, 1:370–392, 1995. ISSN 15710661. doi: 10.1016/S1571-0661(04)00022-2.

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

Melvin E. Conway. A multiprocessor system design. AFIPS '63 (Fall), page 139–146, New York, NY, USA, 1963. Association for Computing Machinery. ISBN 9781450378833. doi: 10.1145/1463822.1463838. URL `https://doi.org/10.1145/1463822.1463838`.

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

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

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multi-threaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP

’95, page 207–216, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917006. doi: 10.1145/209936.209958. URL `https://doi.org/10.1145/209936.209958`.

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

Robert H. Halstead. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP ’84, page 9–17, New York, NY, USA, 1984. Association for Computing Machinery. ISBN 0897911423. doi: 10.1145/800055.802017. URL `https://doi.org/10.1145/800055.802017`.

Umut A Acar. Parallel computing: Theory and practice, 2016. URL `http://www.cs.cmu.edu/afs/cs/academic/class/15210-f15/www/tapp.html`.

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

Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science*, 97(1):1–66, 1992. ISSN 03043975. doi: 10.1016/0304-3975(92)90386-T.

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

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

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

Simon Castellan, Léo Stefanesco, and Nobuko Yoshida. Game semantics: Easy as pi. *CoRR*, abs/2011.05248, 2020.

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: Session-type based reasoning in separation logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–30, 2019.

Zesen Qian, Marco Peressotti, Fabrizio Montesi, and Lars Birkedal. Cell: Concurrent effects in classical linear logic. submitted, 2022.

Pedro Rocha and Luís Caires. Propositions-as-types and shared state. *Proceedings of the ACM on Programming Languages*, 5(ICFP), 2021. doi: 10.1145/3473584.

Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic notes in theoretical computer science*, 319:19–35, 2015.

Tarmo Uustalu. Stateful runners of effectful computations. *Electronic Notes in Theoretical Computer Science*, 319:403–421, 2015.

Matija Pretnar and Gordon D Plotkin. Handling algebraic effects. *Logical methods in computer science*, 9, 2013.

Gordon Plotkin and John Power. Tensors of comodels and models for operational semantics. *Electronic Notes in Theoretical Computer Science*, 218:295–311, 2008.

John Power and Olha Shkaravska. From comodels to coalgebras: State and arrays. *Electronic Notes in Theoretical Computer Science*, 106:297–314, 2004.

Masahito Hasegawa. Linearly Used Effects: Monadic and CPS Transformations into the Linear Lambda Calculus. In Zhenjiang Hu and Mario Rodriguez-Artalejo, editors, *Functional and Logic Programming. FLOPS 2002*, volume 2441 of *Lecture Notes in Computer Science*, pages 167–182, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-44233-2 978-3-540-45788-6. doi: 10.1007/3-540-45788-7_10.

Danel Ahman and Andrej Bauer. Runners in action. In *ESOP*, pages 29–55, 2020.

Paolo Tranquilli. Translating types and effects with state monads and linear logic. 2010.

Dominic Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. *ACM SIGPLAN Notices*, 51(1):568–581, 2016.

Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. Enriching an effect calculus with linear types. In *International Workshop on Computer Science Logic*, pages 240–254. Springer, 2009.

Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. Linearly-used continuations in the enriched effect calculus. In *International Conference on Foundations of Software Science and Computational Structures*, pages 18–32. Springer, 2010.

Rasmus Møgelberg and Sam Staton. Linear usage of state. *Logical Methods in Computer Science*, 10(1):17, March 2014. ISSN 18605974. doi: 10.2168/LMCS-10(1:17)2014.

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. Effective concurrency through algebraic effects. In *OCaml Workshop*, page 13, 2015.

Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. Concurrent system programming with effect handlers. In *International Symposium on Trends in Functional Programming*, pages 98–117. Springer, 2017.

Paulo Emílio de Vilhena and François Pottier. A separation logic for effect handlers. *Proceedings of the ACM on Programming Languages*, 5(POPL): 1–28, 2021.

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. Doo bee doo bee doo. *Journal of Functional Programming*, 30, 2020.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effekt: Lightweight effect polymorphism for handlers. Technical report, Technical Report. University of Tübingen, Germany, 2020a.

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, pages 1–12, 2014.

Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. Syntax and semantics for operations with scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 809–818, 2018.

Zhixuan Yang, Marco Paviotti, Nicolas Wu, Birthe van den Berg, and Tom Schrijvers. Structured handling of scoped effects. In *European Symposium on Programming*, pages 462–491. Springer, Cham, 2022.

Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of logical and algebraic methods in programming*, 84(1): 108–123, 2015.

Daan Leijen. First class dynamic effect handlers: Or, polymorphic heaps with dynamic effect handlers. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development*, pages 51–64, 2018.

NINGNING XIE, YOUYOU CONG, and DAAN LEIJEN. First-class names for effect handlers, 2022.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proceedings of the ACM on Programming Languages*, 4(POPL): 1–29, 2019.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA): 1–30, 2020b.

Yizhou Zhang, Guido Salvaneschi, and Andrew C Myers. Handling bidirectional control flow. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system: Documentation and user's manual. *INRIA*, 3:42, 2022.