Better Late Than Never

A Fully-abstract Semantics for Classical Processes

Kokke, Wen; Montesi, Fabrizio; Peressotti, Marco

Published in:
Proceedings of the ACM on Programming Languages

DOI:
10.1145/3290337

Publication date:
2019

Document version:
Final published version

Better Late Than Never

A Fully-Abstract Semantics for Classical Processes

WEN KOKKE, University of Edinburgh, United Kingdom
FABRIZIO MONTESI, University of Southern Denmark, Denmark
MARCO PERESSOTTI, University of Southern Denmark, Denmark

We present Hypersequent Classical Processes (HCP), a revised interpretation of the "Proofs as Processes" correspondence between linear logic and the π-calculus initially proposed by Abramsky [1994], and later developed by Bellin and Scott [1994], Caires and Pfenning [2010], and Wadler [2014], among others. HCP mends the discrepancies between linear logic and the syntax and observable semantics of parallel composition in the π-calculus, by conservatively extending linear logic to hyperenvironments (collections of environments, inspired by the hypersequents by Avron [1991]). Separation of environments in hyperenvironments is internalised by $\otimes$ and corresponds to parallel process behaviour. Thanks to this property, for the first time we are able to extract a labelled transition system (lts) semantics from proof rewritings. Leveraging the information on parallelism at the level of types, we obtain a logical reconstruction of the delayed actions that Merro and Sangiorgi [2004] formulated to model non-blocking I/O in the π-calculus. We define a denotational semantics for processes based on Brzozowski derivatives, and uncover that non-interference in HCP corresponds to Fubini's theorem of double antiderivation. Having an lts allows us to validate HCP using the standard toolbox of behavioural theory. We instantiate bisimilarity and barbed congruence for HCP, and obtain a full abstraction result: bisimilarity, denotational equivalence, and barbed congruence coincide.

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

Additional Key Words and Phrases: Behavioural Theory, Curry-Howard correspondence, Deadlock-freedom

ACM Reference Format:
Wen Kokke, Fabrizio Montesi, and Marco Peressotti. 2019. Better Late Than Never: A Fully-Abstract Semantics for Classical Processes. *Proc. ACM Program. Lang.* 3, POPL, Article 24 (January 2019), 29 pages. https://doi.org/10.1145/3290337

1 INTRODUCTION

Background. Since its introduction by Girard [1987], linear logic has been tremendously influential in the study of concurrency. Abramsky [1994], and later Bellin and Scott [1994], kickstarted the search for a direct correspondence between proofs in linear logic and processes in (a fragment of) the π-calculus. This direction is appealing because it carries the hope of providing *canonical* foundations for concurrency, ideally as firm as those provided by the Curry-Howard correspondence between natural deduction and the simply-typed λ-calculus for functional programming. These

Authors' addresses: Wen Kokke, Laboratory for Foundations of Computer Science, University of Edinburgh, 10 Crichton Street, Edinburgh, Scotland, EH8 9AB, United Kingdom, wen.kokke@ed.ac.uk; Fabrizio Montesi, Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, Odense, 5230, Denmark, fmontesi@imada.sdu.dk; Marco Peressotti, Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, Odense, 5230, Denmark, peressotti@imada.sdu.dk.

2475-1421/2019/1-ART24
https://doi.org/10.1145/3290337

initial efforts inspired seminal typing disciplines for the π-calculus, *e.g.*, session types by Honda et al. [1998] and linear types by Kobayashi et al. [1999].

Caires and Pfenning [2010] recently revitalised this research line, by developing a correspondence between a variant of the session-typed π-calculus and intuitionistic linear logic: processes correspond to proofs, session types (communication protocols) to propositions, and communication to cut elimination. Wadler [2014] revisited the correspondence for Classical Linear Logic (CLL) and developed the calculus of Classical Processes (CP).

The problem. Despite these recent successes, it is still unclear how we can obtain a unified foundation for concurrency based on linear logic and the π-calculus. This is due to a series of discrepancies between the two theories, both on the levels of syntax and semantics—ultimately, we will see that bridging these gaps leads to a reconciliation of "Proofs as Processes" with the behavioural theory of the π-calculus, in terms of a full abstraction result. As base for our investigation, we use Wadler's calculus CP. CP is convenient to study the discrepancies of interest, because its design is "guided" by linear logic: the syntax of processes in CP corresponds to the structure of the rules of linear logic, and the semantics of these processes is extracted from the traditional steps of cut elimination.

Some discrepancies are syntactic. Parallel composition $P \mid Q$ is a central construct in most process calculi, but only appears combined with output and restriction in CP. The term for output of a linear name in CP is $x[y].(P \mid Q)$, read "send y over x and proceed as P in parallel to Q". Notice that the term constructor for output here actually takes x, y, P and Q as parameters at the same time—whereas in process calculi, only one continuation is typically necessary. This discrepancy is caused by the structure of rule $\otimes$ in CLL, which CP uses to type output: the typing rule checks that the process using y (P) and the one using x (Q) share no resources, by taking two premises (P and Q). In general, there is no independent parallel term $P \mid Q$ in the grammar of CP; Wadler [2014] hints at the possibility of typing $P \mid Q$ using rule MIX by Girard [1987], but this rule does not allow P and Q to synchronise as in the π-calculus. Synchronisation in CP is governed instead by the restriction operator $(\nu xy)(P \mid Q)$, which connects the names x at P and y at Q to enable communication (this restriction term, where x and y represent the two endpoints of a bidirectional channel, was adopted in the latest presentation of CP [Carbone et al. 2016] and was originally introduced by Vasconcelos [2012] for the session-typed π-calculus). Again, parallel is mixed with another operator (restriction), but now it means that P and Q will communicate.

The discrepancies carry over from syntax (and typing) to semantics. Consider the rule for reducing an output with a compatible input in CP, below.

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

Notice how the rule needs to inspect the structure of the continuation of the output term ($P \mid Q$) to produce a typable structure for the resulting network, by nesting restrictions appropriately.

An important consequence of these discrepancies is that CP is still missing a labelled transition system (lts) semantics. Keeping with our example, it is difficult to define a transition axiom for output, as in $x[y].(P \mid Q) \xrightarrow{x[y]} P \mid Q$, because it is not possible to type $P \mid Q$. Even if it were, we hit another problem when attempting to recreate the reduction above using transitions. Ideally, we would define a rule that does not inspect the structure of processes, but only their observables:

$$\frac{P \xrightarrow{x[x']} P' \quad Q \xrightarrow{y'(y')} Q'}{(\nu xy)(P \mid Q) \xrightarrow{\tau} (\nu xy)(\nu x'y')(P' \mid Q')}$$

However, this is not possible because the resulting restriction term is not typable in (nor is in the syntax of) CP. This problem was already noticed by Caires and Pfenning [2010], whose correspondence between τ-transitions and proof normalisation relies on intermediate rewritings that are allowed in the π-calculus, but are not supported by the logic.

Having an lts for CP would be desirable, because it would allow us to study its behavioural theory using the solid toolbox of process calculi based on observable transitions—*e.g.*, bisimilarity (and variations thereof). Also, there is reason to believe that such a study would be interesting. Atkey [2017] informally argued that bisimilarity would be incomplete for CP: for example, CP has no (well-typed) context that can distinguish the processes $x(x').y(y').P$ and $y(y').x(x').P$, since typing would force x and y to be connected to different parallel processes. However, bisimilarity would distinguish these two processes. Motivated by this informal argument, Atkey developed a denotational semantics for processes in CP derived from the relational semantics of linear logic [Barr 1991]. There are still no indications of how this line of work can be reconciled with the standard observational equivalences of process calculi.

Therefore, while the foundations of CP are certainly validated from the side of logic, we are still far from validating them from the side of process calculi.

This paper. We present Hypersequent Classical Processes (HCP), a calculus that mends the discrepancies we discussed between linear logic and the π-calculus. The key twist from linear logic to HCP is to generalise classical linear logic from sequents with single environments to sequents with collections of environments, called hyperenvironments. Hyperenvironments essentially represent independent sequents, inspired by the theory of hypersequents by [Avron 1991], thus the name of HCP. The idea is that whenever two sequents $\vdash \Gamma$ and $\vdash \Delta$ can be proven separately (Γ and Δ are typing environments), then they can be composed as in $\vdash \Gamma \mid \Delta$, where $\Gamma \mid \Delta$ is a hyperenvironment. Intuitively, each environment in a hyperenvironment can be proven independently—in parallel, if you like. From a logical perspective, the operator "$\mid$" for composing hyperenvironments is internalised by the $\otimes$ connective of linear logic (if $\vdash \Gamma, A \mid \Delta, B$, then $\vdash \Gamma, \Delta, A \otimes B$), just like "," for composing environments is internalised by the $\parr$ connective (if $\vdash \Gamma, A, B$, then $\vdash \Gamma, A \parr B$). This new symmetric treatment of $\otimes$ and $\parr$ is the foundation of all our contributions, which we believe represent a concrete step forward in Abramsky's original programme of "Proofs as Processes". Our first contributions deal with the design of HCP, whereas the others with its validation. Best comes last: our entire development is validated by the titular result of this paper, a full abstraction result that ties together bisimilarity, denotational equivalence, and contextual equivalence for HCP.

(1) HCP reconciles the structure of proofs with the syntax of processes. On the process calculus side, term constructors have the expected modularity of process algebras, *e.g.*, parallel composition and restriction are respectively the usual abelian monoid $P \mid Q$ and the term $(\nu xy)P$ of the session-typed π-calculus [Vasconcelos 2012]. We formalise that HCP is grounded in classical linear logic (CLL) by proving that the two systems are equally powerful: we can internalise the new ingredient of environment composition using the connective $\otimes$.
(2) HCP supports sound proof rewritings that correspond to transition rules for processes, which we use to extract an lts. Our lts mends the discrepancy we discussed about semantics, and extends the Curry-Howard correspondence of "Proofs as Processes" to the SOS style by Plotkin [2004], by viewing proofs as states and our new proof rewritings as transitions. Well-typed processes enjoy progress in our lts.
(3) Thanks to the fact that hyperenvironments allow us to see independence at the level of types (the "$\mid$" operator for composing environments), HCP supports new proof rewritings that yield a logical reconstruction of the lts originally studied by Merro and Sangiorgi [2004] for the π-calculus with non-blocking I/O (delayed actions).
(4) Our lts bridges the gap between the research lines of "Proofs as Processes" and of behavioural theory for process calculi. As the first step on this bridge, we instantiate standard bisimilarity for HCP. Bisimilarity gives us two immediate confirmations that our lts is sound: well-typed processes that are bisimilar are also type equivalent; and bisimilarity is a congruence.

Courtesy of delayed actions, bisimilarity relates Atkey's problematic processes $x(x').y(y').P$ and $y(y').x(x').P$. Even further, bisimilarity characterises (coincides with) contextual equivalence (for HCP, this is typed barbed congruence). While the completeness of bisimilarity is not a requirement, it is certainly desirable—and somewhat expected, for a first-order process calculus [Sangiorgi and Walker 2001].

(5) We define a denotational semantics for HCP, by reformulating the one for CP by Atkey [2017]. Atkey's denotations are inspired by the relational semantics of CLL by Barr [1991]. We rediscover (a refinement of) these denotations from a different angle, by defining Brzozowski derivatives [Brzozowski 1964] w.r.t. the observable actions in our lts. This has three benefits. First, it gives a formal and direct connection between the operational and denotational semantics of HCP. Second, it shows that the denotational semantics of HCP agrees with a standard notion of observability. Third, it reveals that non-interference, usually a topic of operational semantics, can be stated for HCP in denotational terms: Fubini's theorem of double antiderivation holds in our setting [Fubini 1907], formalising the intuition that the order of independent actions is not discriminated. In a sense, Fubini's theorem for HCP explains from a denotational perspective why delayed actions are operationally sound.

(6) As we anticipated, HCP enjoys full abstraction, in the sense that all three semantic equivalences we present coincide: bisimilarity = denotational = contextual equivalence.

Wadler [2014] ended his presentation of Classical Processes by stating:

> "As λ-calculus provided foundations for functional programming in the last century, may we hope for this emerging calculus to provide foundations for concurrent programming in the coming century?"

From the riverbank of behavioural theory for process calculi, delaying the execution of actions seems to be an important aspect for this agenda. Better late than never.

2 HYPERSEQUENT CLASSICAL PROCESSES

We start our formal development by presenting the process syntax and proof theory of Hypersequent Classical Processes (HCP).

2.1 Processes

In HCP, programs are processes (P,Q,R,…) that communicate using names (x,y,z,…). A name represents one of the two endpoints of a bidirectional channel. This style was introduced to the session-typed π-calculus by Vasconcelos [2012], and later adopted in the latest presentation of Classical Processes by Carbone et al. [2016]. Process terms are given by the following grammar.

$P, Q ::=$	$x[y].P$	*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*
$\mid$	$x \triangleleft \mathsf{inl}.P$	*select left on x and continue as P*
$\mid$	$x \triangleleft \mathsf{inr}.P$	*select right on x and continue as P*
$\mid$	$x \triangleright \{\mathsf{inl} : P; \mathsf{inr} : Q\}$	*offer a binary choice between P (left) or Q (right) on x*
$\mid$	$!x(y).P$	*offer a service*
$\mid$	$?x[y].P$	*consume a service*
$\mid$	$?x[x_1, x_2].P$	*duplicate a service*
$\mid$	$?x[].P$	*dispose of a service*
$\mid$	$(\nu xy)P$	*name restriction, "cut"*

|	$P \mid Q$	*parallel composition of processes P and Q*
|	$\mathbf{0}$	*terminated process*
|	$x{\leftrightarrow}y$	*link x and y*

Term $x[y].P$ allocates a fresh name y, outputs y over x and then proceeds as P. Dually, term $x(y).P$ inputs a name y over x and then proceeds as P. Both output and input terms bind the transmitted name (y) to the respective continuation P. Terms $x[].P$ and $x().P$ respectively model output and input with no content. Terms $x \triangleleft \mathsf{inl}.P$ and $x \triangleleft \mathsf{inr}.P$ respectively send on x the selection of the left or right branch of a (binary) offer available on the other end of the channel before proceeding as P. Dually, term $x \triangleright \{\mathsf{inl} : P; \mathsf{inr} : Q\}$ offers on x a choice between proceeding as P (left branch) or Q (right branch). Term $!x(y).P$ is a server that offers on x a service implemented by the replicable process P, where y is bound in P. A server term can be used by clients any number of times. Accordingly, we have three client terms to interact with a server. The client term $?x[y].P$ requests exactly one copy of the service provided by the server on x, and then proceeds by communicating with the service on channel y. The client term $?x[].P$ disposes the server on x—the service is used zero times. The client term $?x[x_1, x_2].P$ requests that the server on x is duplicated in two new instances, respectively available on the new channels x_1 and x_2. A restriction term $(\nu xy)P$ forms a channel by connecting and binding the two endpoints x and y in P, enabling communications from x to y and *vice versa*. Restriction hides the endpoints x and y from the context. Terms $P \mid Q$ and $\mathbf{0}$ are the standard terms for the parallel composition of two processes and the terminated process. Term $x{\leftrightarrow}y$ is a forwarding proxy: inputs on x are forwarded as outputs on y and *vice versa*.

In the remainder, we use π to range over term prefixes: $x[y]$, $x(y)$, $x[]$, $x()$, $x \triangleleft \mathsf{inl}$, $x \triangleleft \mathsf{inr}$, $!x(y)$, $?x[y]$, $?x[x_1, x_2]$, and $?x[]$. Free and bound names of processes and prefixes are defined as expected, as well as α-conversion. We write $\mathrm{fn}(P)$, $\mathrm{bn}(P)$, $\mathrm{cn}(P)$, for the set of free, bound, and all channel names in P, respectively, and likewise for prefixes. We write $P =_\alpha Q$ if P and Q are α-equivalent.

Example 2.1. We write a server that computes the logical AND of two bits, adapting an example by Atkey et al. [2016] to HCP. We use selections to model sending bits. Since HCP is pretty low-level as a programming language, we use the following syntactic sugar.

$$x[0].P \triangleq x[x'].x' \triangleleft \mathsf{inl}.x'[].P \qquad x[1].P \triangleq x[x'].x' \triangleleft \mathsf{inr}.x'[].P$$
$$x \triangleright \{0 \mapsto P; 1 \mapsto Q\} \triangleq x \triangleright \{\mathsf{inl} : x().P; \mathsf{inr} : x().Q\}$$

With these abbreviations, we can write a server that offers a service for computing logical AND.

$$\mathit{Server}_y \triangleq !y(y').y'(p).y'(q).p \triangleright \begin{Bmatrix} 0 \mapsto q \triangleright \{0 \mapsto y'[0].y'[].\mathbf{0}; 1 \mapsto y'[0].y'[].\mathbf{0}\} \\ 1 \mapsto q \triangleright \{0 \mapsto y'[0].y'[].\mathbf{0}; 1 \mapsto y'[1].y'[].\mathbf{0}\} \end{Bmatrix}$$

We now define a compatible client, $\mathit{Client}_{xz}^{b_1 b_2}$, which sends bits b_1 and b_2 (0 or 1) to a server that accepts two bits on x (the client abstracts from the concrete operation that the server computes). The client uses the result to decide whether to select left or right on another channel z.

$$\mathit{Client}_{xz}^{b_1 b_2} = ?x[x'].x'[b_1].x'[b_2].x' \triangleright \{0 \mapsto x'().z \triangleleft \mathsf{inl}.z[].\mathbf{0}; 1 \mapsto x'().z \triangleleft \mathsf{inr}.z[].\mathbf{0}\}$$

Relation to other calculi. The main difference between the syntax of HCP and its predecessors in the research line of "Proofs as Processes" is that parallel composition $P \mid Q$ is a term in its own right instead of being an inseparable subcomponent of other terms, as we discussed in the Introduction. Our restriction and output terms have the familiar arities of the π-calculus: output $x[y].P$ has a single continuation, and likewise restriction $(\nu xy)P$ binds xy to a single process (instead of two). Of course, designing an "expected" syntax for a session-typed process calculus is not hard—otherwise, it would not be expected! The real challenge is designing a proof theory based on linear logic where the structures of proofs match this syntax precisely, as we will do in Section 2.2.

Our client terms for explicit server management are inspired by Wadler [2014], who presented them as an alternative notation for Classical Processes (CP). In CP, server duplication and disposal do not have terms: these actions are handled by the semantics of CP by looking at the typing proofs of processes. We chose the explicit terms for HCP because, as we will see, server duplication and disposal are *observable* actions. Thus, to define an lts in the usual SOS style, it is desirable that these observables arise from corresponding syntactic terms.

From the perspective of π-calculus, HCP is essentially a fragment of the internal π-calculus by Sangiorgi [1996], with two differences. First, the explicit management of servers (our client terms, which we will see correspond to the rules for the exponential connective "?"). Second, the fact that channels are formed explicitly by the restriction term as proposed later by Vasconcelos [2012], rather than implicitly by using the same name in different processes. The hallmark of the internal π-calculus is that output always sends a fresh name, as in HCP. This makes the theory of the calculus more convenient (output and input are symmetrical). The usual π-calculus term for outputting a free name can be recovered as syntactic sugar by using links [Atkey et al. 2016].

$$x\langle y\rangle.P \triangleq x[z].(y{\leftrightarrow}z \mid P)$$

Similar considerations apply to polyadic communications [Sangiorgi and Walker 2001].

2.2 Typing

Types. HCP uses propositions from Classical Linear Logic (CLL) as types for (endpoint) names. Types (A,B,C,…) are defined by the following grammar.

$A, B ::=$	$A \otimes B$	*send A, proceed as B*	$\mid$	$A \parr B$	*receive A, proceed as B*
$\mid$	$A \oplus B$	*select A or B*	$\mid$	$A \mathbin{\&} B$	*offer A or B*
$\mid$	1	*unit for* $\otimes$	$\mid$	$\bot$	*unit for* $\parr$
$\mid$	$?A$	*client request*	$\mid$	$!A$	*server accept*

Types on the left-hand column are for outputs and types in the right-hand column for inputs. Connectives on the same row are respective duals, *e.g.*, $\otimes$ and $\parr$ are dual of each other. We assume the standard notion of duality of CLL, writing $A^\bot$ for the dual of A. Duality proceeds homomorphically and replaces connectives with their duals, for example $(A \otimes B)^\bot = A^\bot \parr B^\bot$.

Environments and Hyperenvironments. Let Γ, Δ, Θ range over unordered environments, which associate names to types.

$$\Gamma, \Delta, \Theta ::= x_1 : A_1, \ldots, x_n : A_n \quad \textit{environment}$$

We write $\bullet$ for the empty environment. Given an environment $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, we write $\mathrm{cn}(\Gamma)$ for the set $\{x_1, \ldots, x_n\}$ of names in Γ. Names in the same environment must be distinct. Two environments can be composed only if they do not share names: whenever we write Γ, Δ, this implies $\mathrm{cn}(\Gamma) \cap \mathrm{cn}(\Delta) = \emptyset$.

Environments are collected in unordered hyperenvironments, ranged over by $\mathcal{G}, \mathcal{H}$.

$$\mathcal{G}, \mathcal{H} ::= \Gamma_1 \mid \cdots \mid \Gamma_n \quad \textit{hyperenvironment}$$

The idea is that all environments in a hyperenvironment can be proven independently. We write $\varnothing$ for the empty hyperenvironment and $\mathrm{cn}(\mathcal{H})$ for the set of names appearing in (all the environments in) $\mathcal{H}$. As for environments, we require all names in hyperenvironments to be distinct: $\mathcal{G} \mid \mathcal{H}$ implies $\mathrm{cn}(\mathcal{G}) \cap \mathrm{cn}(\mathcal{H}) = \emptyset$. Environments and hyperenvironments are equated up to exchange: $\Gamma, \Delta = \Delta, \Gamma$ and $\mathcal{G} \mid \mathcal{H} = \mathcal{H} \mid \mathcal{G}$.

Structural rules

$$\frac{}{x{\leftrightarrow}y \vdash x : A^{\perp}, y : A}\,\text{AX} \qquad \frac{P \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : A^{\perp}}{(\nu xy)P \vdash \mathcal{G} \mid \Gamma, \Delta}\,\text{H-CUT} \qquad \frac{P \vdash \mathcal{G} \quad Q \vdash \mathcal{H}}{P \mid Q \vdash \mathcal{G} \mid \mathcal{H}}\,\text{H-MIX} \qquad \frac{}{\mathbf{0} \vdash \varnothing}\,\text{H-MIX}_0$$

Logical rules

$$\frac{P \vdash \mathcal{G} \mid \Gamma, y : A \mid \Delta, x : B}{x[y].P \vdash \mathcal{G} \mid \Gamma, \Delta, x : A \otimes B}\,\otimes \qquad \frac{P \vdash \mathcal{G}}{x[].P \vdash \mathcal{G} \mid x : 1}\,1 \qquad \frac{P \vdash \mathcal{G} \mid \Gamma, y : A, x : B}{x(y).P \vdash \mathcal{G} \mid \Gamma, x : A \parr B}\,\parr \qquad \frac{P \vdash \mathcal{G} \mid \Gamma}{x().P \vdash \mathcal{G} \mid \Gamma, x : \perp}\,\perp$$

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x : A}{x \triangleleft \mathsf{inl}.P \vdash \mathcal{G} \mid \Gamma, x : A \oplus B}\,\oplus_1 \qquad \frac{P \vdash \mathcal{G} \mid \Gamma, x : B}{x \triangleleft \mathsf{inr}.P \vdash \mathcal{G} \mid \Gamma, x : A \oplus B}\,\oplus_2 \qquad \frac{P \vdash \Gamma, x : A \quad Q \vdash \Gamma, x : B}{x \triangleright \{\mathsf{inl} : P; \mathsf{inr} : Q\} \vdash \Gamma, x : A \,\&\, B}\,\&$$

$$\frac{P \vdash ?\Gamma, y : A}{!x(y).P \vdash ?\Gamma, x : !A}\,! \qquad \frac{P \vdash \mathcal{G} \mid \Gamma, y : A}{?x[y].P \vdash \mathcal{G} \mid \Gamma, x : ?A}\,? \qquad \frac{P \vdash \mathcal{G} \mid \Gamma}{?x[].P \vdash \mathcal{G} \mid \Gamma, x : ?A}\,\text{W} \qquad \frac{P \vdash \mathcal{G} \mid \Gamma, x' : ?A, x'' : ?A}{?x[x', x''].P \vdash \mathcal{G} \mid \Gamma, x : ?A}\,\text{C}$$

Fig. 1. HCP, typing rules.

Judgements and Typing. Typing judgements assign processes to hyperenvironments and have the form: $P \vdash \mathcal{G}$. The rules for deriving judgements are displayed in Figure 1. We say that a process P is *well-typed* whenever there exists some $\mathcal{G}$ such that $P \vdash \mathcal{G}$.

Remark 2.2 (Alternative notation). An alternative notation for our judgements could be $P :: \vdash \Gamma_1 \mid \cdots \mid \vdash \Gamma_n$ because, as we will show later, each sequent $\vdash \Gamma_i$ is always guaranteed to be independently provable in classical linear logic. Thus, our judgements can be seen as collections of sequents, recalling the hypersequents by Avron [1991]. This is the reason behind the name of HCP. We chose our notation to reduce eyestrain.

Typing rules associate types to names by looking at how endpoints are used in process terms. Rule selection is structural on the syntax of processes, in the sense that it depends only on the outermost constructor of a process term. In rule !, we write ?Γ for an environment of the form $?A_1, \ldots, ?A_n$ (possibly empty).

Most of our rules—with the exception of H-CUT (restriction), ⊗ (output), H-MIX (parallel composition), and H-MIX_0 (terminated process)—are exactly those presented for Classical Linear Logic (CLL) by Girard [1987], but extended to hyperenvironments. Dual terms are typed with dual types.

The most important new rules are the structural rules H-MIX and H-CUT. Rule H-MIX types the parallel composition of two processes, by combining their hyperenvironments. Previous work proposed a different rule for mixing environments, given below [Girard 1987; Wadler 2014].

$$\frac{P \vdash \Gamma \quad Q \vdash \Delta}{P \mid Q \vdash \Gamma, \Delta}\,\text{MIX}$$

Notice the key difference: our rule keeps the information that the resources in the two premises come from independent proofs. This information allows us to reformulate cut as rule H-CUT, which uses a single premise. Rule H-CUT types the a restriction $(\nu xy)P$ by checking that the channel is used by parallel components (separate environments) in P in a dual way (as usual in CLL). In general, the key novelty of HCP is that parallelism is guaranteed by separation of hyperenvironments. By contrast, the standard cut rule of linear logic requires two separate *proofs* as premises, yielding the restriction term constructor $(\nu x)(P \mid Q)$ that we discussed in the Introduction. Rules H-MIX and H-CUT and hyperenvironments form thus the key to the desired decoupling of restriction and the parallel operator. Rule H-MIX_0 types $\mathbf{0}$, the unit of parallel composition for processes, as $\varnothing$, the unit of composition for hyperenvironments.

Our rule ⊗ is reformulated from CLL using the same intuition for rule H-CUT. The original rule requires two separate proofs for A and B respectively, whereas ours has a single premise requiring that A and B are in separate environments. In other words, ⊗ internalises | in propositions, which yields a logical reconstruction of the output term from the internal π-calculus [Sangiorgi 1996].

The other rules are straightforward adaptations to hyperenvironments of the rules in [Wadler 2014] for Classical Processes. Rule AX types a link (forwarder), checking that the connected endpoints have dual types. This ensures that any message on x can be safely forwarded to y, and *vice versa*. All logical rules enforce linear usage, except for client requests (typed with the exponential connective ?), for which contraction and weakening are allowed. Contraction (rule C) allows for multiple client requests for the same server, and weakening (rule W) for not using a server.

Types are preserved under α-conversion, in the sense that whenever $P =_\alpha Q$, $P \vdash \mathcal{G}$ iff $Q \vdash \mathcal{G}$.

Example 2.3. Define the types for sending and receiving a bit, respectively.

$$Bit = 1 \oplus 1 \quad \textit{send a bit} \qquad\qquad Bit^\perp = \bot \,\&\, \bot \quad \textit{receive a bit}$$

Then, we can type the server and client terms from Example 2.1 with dual types, as follows.

$$Server_y \vdash y : !(Bit^\perp \parr Bit^\perp \parr Bit \otimes 1) \qquad Client_{xz}^{b_1 b_2} \vdash x : ?(Bit \otimes Bit \otimes Bit^\perp \parr \bot), z : Bit$$

Thus, by rules H-CUT and H-MIX we can type their composition for all distinct names x, y and z, and any bits b_1 and b_2, *e.g.*, to compute the logical AND of 0 and 1: $(\nu xy)\left(Client_{xz}^{01} \mid Server_y\right) \vdash z : Bit$.

For some processes, there are different acceptable ways of distributing free names in hyperenvironments. For example, the process $x().y[].z[].\mathbf{0}$ is typed by both $x : \bot, y : 1 \mid z : 1$ and $y : 1 \mid x : \bot, z : 1$, the only difference being that the name x appears in a different component (environment). In general, given $P \vdash \mathcal{G}$, for any P and $\mathcal{G}$, if we erase types from $\mathcal{G}$ then we obtain a partition of the free names of P. We write $\lfloor \mathcal{G} \rfloor$ for the name partition obtained by removing types from $\mathcal{G}$. For example, $\lfloor x : \bot, y : 1 \mid z : 1 \rfloor$ is $x, y \mid z$ (corresponding to $\{\{x, y\}, \{z\}\}$ in standard set notation). Intuitively, name partitions describe which names are used by each parallel component of a process. We write a judgement $P \Vdash G$ to say that a process P supports the partition G on the set of its free names. The rules for deriving partitioning judgements are obtained by erasing all types (A, B, and connectives) from the typing rules displayed in Figure 1 and replacing "$\vdash$" with "$\Vdash$" (we omit these rules for conciseness). Thus, name partitions are independent of typing. Computing all the possible name partitions for a process is trivially decidable: the set of free names of a process is always finite, giving a bound on the number of possible partitions. Any derivation for $P \vdash \mathcal{G}$ is also a derivation for $P \Vdash \lfloor \mathcal{G} \rfloor$ once we erase channel types but not *vice versa*: just consider $P = x[].\mathbf{0}$ and $\mathcal{G} = x : 1 \oplus 1$ (thus $\lfloor \mathcal{G} \rfloor = x$). We write $\mathrm{np}(P)$ for the set $\{G \mid P \Vdash G\}$ of name partitions induced by P.

We say that two hyperenvironments $\mathcal{G}$ and $\mathcal{G}'$ are one the *shuffling* of the other, written $\mathcal{G} \sqcup\!\sqcup \mathcal{G}'$, whenever they count the same number of non-empty environments and $x : A$ is in $\mathcal{G}$ iff $x : A$ is in $\mathcal{G}'$.

THEOREM 2.4. *If $P \vdash \mathcal{G}$, $P \Vdash \lfloor \mathcal{G}' \rfloor$, and $\mathcal{G} \sqcup\!\sqcup \mathcal{G}'$ then, $P \vdash \mathcal{G}'$.*

2.3 Relation with Classical Linear Logic

If we erase processes and names from our typing rules and judgements, we essentially get a linear proof theory and sequents based on hyperenvironments. We write $\vdash \mathcal{G}$ when working under this erasure, abusing notation ($\mathcal{G}$ does not contain names in this case).

We root HCP in CLL by relating their proof theories. We start from the easier direction: all proofs in CLL can be encoded into proofs in HCP. Intuitively, this is because all rules in CLL but CUT and ⊗ are present also in HCP (taking $\mathcal{G}$ as empty). It is straightforward to reconstruct the missing rules by combining H-MIX with ⊗ and H-CUT.

Theorem 2.5. *If* $\vdash \Gamma$ *in* CLL *then* $\vdash \Gamma$ *in* HCP.

If we consider processes, from the proof of Theorem 2.5 we extract the expected encoding from the latest version of Wadler's Classical Processes (CP, which uses CLL as typing discipline) by Carbone et al. [2016] to visually identical terms in HCP, *e.g.*, $[\![(\nu xy)(P \mid Q)]\!] = (\nu xy)([\![P]\!] \mid [\![Q]\!])$. This means that all well-typed processes in CP are well-typed also in HCP.

The opposite direction, from HCP to CLL, is not as straightforward because HCP supports proof structures that do not appear in CLL. From a process perspective, there are behaviours that cannot be translated directly from HCP to CP. For example, the process $x \triangleright \{\mathsf{inl} : P \mid Q; \mathsf{inr} : P' \mid Q'\}$, where x appears in Q and Q', is typable in HCP but cannot be written/typed in CP. The choice sent on x will affect the choice between P and P', even though neither has access to x.

Instead, we will prove that HCP supports the same propositions as CLL. This is the same as saying that HCP and CP inhabit the same types, or that the associated logical systems derive the same theorems. We use a standard method for proving the soundness of hypersequent calculi: hyperenvironments are internalised as propositions in CLL.

We observe that all proofs in HCP can be "disentangled", by moving applications of rule H-MIX deeper in the proof tree. We can use this property to rewrite any derivation to a form in which all mixes are either attached to their respective cuts or tensors, or at the top-level. These consecutive applications can be rewritten as rule applications of cut and $\otimes$ from CLL.

Lemma 2.6 (Disentanglement). *If there exists a derivation* ρ *of* $\vdash \Gamma_1 \mid \cdots \mid \Gamma_n$ *in* HCP*, then there exist derivations* $\rho_1, \ldots, \rho_n$ *of* $\vdash \Gamma_1, \ldots, \vdash \Gamma_n$ *in* CLL.

We define an encoding of hyperenvironments in HCP into propositions in CLL.

$$\parr(\bullet) = \bot \qquad \bigotimes(\varnothing) = 1 \qquad \parr(\Gamma, A) = \parr(\Gamma) \parr A \qquad \bigotimes(\mathcal{G} \mid \Gamma) = \bigotimes(\mathcal{G}) \otimes \bigotimes(\Gamma)$$

Lemma 2.7. *If* $\vdash \Gamma$ *in* HCP*, then* $\vdash \parr \Gamma$ *in* CLL.

By Lemma 2.7 and repeated applications of $\otimes$ in CLL, we obtain the following theorem.

Theorem 2.8. *If* $\vdash \mathcal{G}$ *in* HCP*, then* $\vdash \bigotimes \mathcal{G}$ *in* CLL.

3 OPERATIONAL SEMANTICS

HCP supports new proof rewritings w.r.t. CLL, which correspond to transition rules for processes. We use this property to define a semantics for HCP in terms of a labelled transition system (lts). Our semantics follows Plotkin's SOS style [Plotkin 2004], by viewing:

- the inference rules of our type system as operations of a (sorted) signature;
- proofs as terms generated by this signature;
- (labelled) proof transformations as (labelled) transitions;
- and a specification of rules for deriving proof transformations as an SOS specification.

Then, a semantics for HCP processes in terms of an SOS specification is obtained simply by reading off how the SOS specification of proof transformations manipulate the processes that they type.

To illustrate the intuition for transitions, consider the proof for a judgement $x().P \vdash \mathcal{G} \mid \Gamma, x : \bot$. By the correspondence between term constructors and typing rules, the proof has the following shape.

$$\dfrac{\begin{array}{c}\vdots\\ P \vdash \mathcal{G} \mid \Gamma\end{array}}{x().P \vdash \mathcal{G} \mid \Gamma, x : \bot}\ \bot$$

We can view rule $\bot$ as the outermost operation used in the proof. Then, the proof of $P \vdash \mathcal{G} \mid \Gamma$ is the only argument of the operation and x a parameter (operations are on proofs). This corresponds to the term constructor $x().(-)$ in the syntax of HCP processes—which in this case takes P as

Action labels (l, l', …)

$x[]$ close x	$x()$ wait on x	$x[y]$ output y on x	$x(y)$ input y on x
$x \triangleleft \mathsf{inl}$ select left	$x \triangleright \mathsf{inl}$ offer left	$x \triangleleft \mathsf{inr}$ select right	$x \triangleright \mathsf{inr}$ offer right
$?x[y]$ request y on x	$?x[]$ request dispose x	$?x[x_1, x_2]$ request duplicate x	$x \leftrightarrow y$ forward
$!x(y)$ accept y on x	$!x()$ accept dispose x	$!x(x_1, x_2)$ accept duplicate x	

Actions

$$\frac{\pi \neq ?x[], ?x[x_1, x_2]}{\pi.P \xrightarrow{\pi} P}\,\pi \qquad ?x[].P \xrightarrow{?x[]} x().P \qquad ?x[x_1, x_2].P \xrightarrow{?x[x_1,x_2]} x_1(x_2).P$$

$$x \triangleright \{\mathsf{inl} : P; \mathsf{inr} : Q\} \xrightarrow{x \triangleright \mathsf{inl}} P \quad x \leftrightarrow y \xrightarrow{x \leftrightarrow y} \mathbf{0} \qquad x \triangleright \{\mathsf{inl} : P; \mathsf{inr} : Q\} \xrightarrow{x \triangleright \mathsf{inr}} Q \quad x \leftrightarrow y \xrightarrow{y \leftrightarrow x} \mathbf{0}$$

$$\frac{\mathrm{fn}(P) = \{x', z_1, \ldots, z_n\}}{!x(x').P \xrightarrow{!x()} ?z_1[]. \cdots ?z_n[].x[].\mathbf{0}}\,\text{DISP}$$

$$\frac{P_1 = P\sigma_1 \quad P_2 = P\sigma_2 \quad \mathrm{fn}(P_1) \cap \mathrm{fn}(P_2) = \emptyset \quad \mathrm{fn}(P) = \{x', z_1, \ldots, z_n\}}{!x(x').P \xrightarrow{!x(x_1,x_2)} ?z_1[z_1\sigma_1, z_1\sigma_2]. \ldots ?z_n[z_n\sigma_1, z_n\sigma_2].x_1[x_2].(!x_1(x'\sigma_1).P_1 \mid !x_2(x'\sigma_2).P_2)}\,\text{DUP}$$

Structural

$$\frac{P \xrightarrow{l} P' \quad \mathrm{bn}(l) \cap \mathrm{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{l} P' \mid Q}\,\text{PAR}_1 \qquad \frac{Q \xrightarrow{l} Q' \quad \mathrm{bn}(l) \cap \mathrm{fn}(P) = \emptyset}{P \mid Q \xrightarrow{l} P \mid Q'}\,\text{PAR}_2 \qquad \frac{P \xrightarrow{y \leftrightarrow z} P'}{(\nu xy)P \xrightarrow{\tau} P'\{x/z\}}\,\text{AXCUT}$$

$$\frac{P \xrightarrow{l} P' \quad Q \xrightarrow{l'} Q' \quad \mathrm{bn}(l) \cap \mathrm{bn}(l') = \emptyset}{P \mid Q \xrightarrow{(l \| l')} P' \mid Q'}\,\text{SYN} \qquad \frac{P \xrightarrow{l} P' \quad x, y \notin \mathrm{cn}(l) \quad x *_{P'} y}{(\nu xy)P \xrightarrow{l} (\nu xy)P'}\,\text{RES} \qquad \frac{P =_\alpha Q \quad Q \xrightarrow{l} R}{P \xrightarrow{l} R}\,=_\alpha$$

Communications

$$\frac{P \xrightarrow{(x[x'] \| y(y'))} P'}{(\nu xy)P \xrightarrow{\tau} (\nu xy)(\nu x'y')P'}\,\otimes\parr \qquad \frac{P \xrightarrow{(x \triangleleft \mathsf{inl} \| y \triangleright \mathsf{inl})} P'}{(\nu xy)P \xrightarrow{\tau} (\nu xy)P'}\,\oplus_1\& \qquad \frac{P \xrightarrow{(x \triangleleft \mathsf{inr} \| y \triangleright \mathsf{inr})} P'}{(\nu xy)P \xrightarrow{\tau} (\nu xy)P'}\,\oplus_2\&$$

$$\frac{P \xrightarrow{(x[] \| y())} P'}{(\nu xy)P \xrightarrow{\tau} P'}\,1\bot \qquad \frac{P \xrightarrow{(?x[x'] \| !y(y'))} P'}{(\nu xy)P \xrightarrow{\tau} (\nu x'y')P'}\,!? \qquad \frac{P \xrightarrow{(?x[] \| !y())} P'}{(\nu xy)P \xrightarrow{\tau} (\nu xy)P'}\,!\mathrm{W} \qquad \frac{P \xrightarrow{(?x[x_1,x_2] \| !y(y_1,y_2))} P'}{(\nu xy)P \xrightarrow{\tau} (\nu x_1y_1)P'}\,!\mathrm{C}$$

Delayed Actions and Self-synchronisations

$$\frac{\pi \in \{x[], x \triangleleft \mathsf{inl}, x \triangleleft \mathsf{inr}, ?x[y]\} \quad P \xrightarrow{l} P' \quad \mathrm{cn}(\pi) \cap \mathrm{cn}(l) = \emptyset}{\pi.P \xrightarrow{l} \pi.P'}\,:\pi_1 \qquad \frac{\pi \in \{x(), ?x[]\} \quad P \xrightarrow{l} P' \quad \mathrm{fn}(P') \neq \emptyset}{\pi.P \xrightarrow{l} \pi.P'}\,:\pi_2$$

$$\frac{P \xrightarrow{l} P' \quad x, x' \notin \mathrm{cn}(l) \quad x *_{P'} x'}{x[x'].P \xrightarrow{l} x[x'].P'}\,:\otimes \qquad \frac{\pi \in \{x_1(x_2), ?x[x_1, x_2]\} \quad P \xrightarrow{l} P' \quad \mathrm{cn}(\pi) \cap \mathrm{cn}(l) = \emptyset \quad x_1 \circledast_{P'} x_2}{\pi.P \xrightarrow{l} \pi.P'}\,:\pi_3$$

$$\frac{\pi \neq ?x[], ?x[x_1, x_2] \quad \pi.P \xrightarrow{l} \pi.P' \quad \mathrm{fn}(\pi) *_{\pi.P} \mathrm{fn}(l)}{\pi.P \xrightarrow{(\pi \| l)} P'}\,|\pi \qquad \frac{?x[].P \xrightarrow{l} ?x[].P' \quad x *_{?x[].P} \mathrm{fn}(l)}{?x[].P \xrightarrow{(?x[] \| l)} x().P'}\,|\mathrm{W}$$

$$\frac{?x[x_1, x_2].P \xrightarrow{l} ?x[x_1, x_2].P' \quad x *_{?x[x_1,x_2].P} \mathrm{fn}(l)}{?x[x_1, x_2].P \xrightarrow{(?x[x_1,x_2] \| l)} x_1(x_2).P'}\,|\mathrm{C}$$

Fig. 2. Labelled transition system of HCP processes.

argument, *i.e.*, the term corresponding to the proof of the premise. Thus, this operation is the proof equivalent of the term constructor $x().(-)$ in the syntax of HCP processes, which denotes an observable action. Term constructors like this, also called action prefixes, are typically assigned

a transition rule in process calculi where the target (a.k.a. *derivative*) is the operator argument and the label is the prefixed operation. This correspondence points at the transition rule below—for readability, we box proofs and omit proof trees above premises in the remainder.

$$\frac{P \vdash \mathcal{G} \mid \Gamma}{x().P \vdash \mathcal{G} \mid \Gamma, x : \bot}\,\bot \xrightarrow{x() : \bot} P \vdash \mathcal{G} \mid \Gamma$$

The label identifies the prefix constructor (*i.e.*, rule name and parameter) and its syntax is inspired by common syntax for labels of action prefixes in process calculi. By reading proof terms (processes) off the rule above we obtain the axiom below for processes.

$$x().P \xrightarrow{x()} P$$

This axiom defines the expected semantics of the constructor $x().(-)$, a promising sign!

Following this methodology for all of our typing rules, we obtain the lts on HCP processes given by the SOS specification in Figure 2, where l ranges over transition labels (we abuse notation and use l for the red part of transition labels when referring to processes only, and both the red and blue parts when referring to proof transitions). We describe each transition rule in the remainder of this section, by discussing the proof transformations that they originate from. In transitions, we identify α-equivalent processes.

In the sequel we write $x *_P y$ (resp. $x \circledast_P y$) whenever there is a partition $G \in \mathrm{np}(P)$ that separates (resp. does not separate) x and y. We write $x *_P y_1, \ldots, y_n$ for $\bigwedge_{i=1}^{n} x *_P y_i$.

3.1 Multiplicatives and Mix

We start by giving a semantics to the multiplicative fragment of HCP, which suffices to show all the key ideas behind our lts. The multiplicative fragment of HCP is formed by the rules $\otimes$, $\parr$, 1 and $\bot$, together with the structural rules H-MIX, H-MIX$_0$ and H-CUT. Observe that rules from the first group have the "action prefix" form described above.

Actions. The transition rules for multiplicative prefixes are those below, plus the rule for $x().P$ already given.

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

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x' : A, x : B}{x(x').P \vdash \mathcal{G} \mid \Gamma, x : A \parr B}\,\parr \xrightarrow{x(x') : A \parr B} P \vdash \mathcal{G} \mid \Gamma, x' : A, x : B \qquad \frac{P \vdash \mathcal{G}}{x[].P \vdash \mathcal{G} \mid x : 1}\,1 \xrightarrow{x[] : 1} P \vdash \mathcal{G}$$

Structural rules. There are three transition rules for rule H-MIX: two for executions where only one component is transformed (rules PAR$_1$ and PAR$_2$) and one where both components are transformed synchronously (rule SYN). (We omit rule PAR$_2$, which is symmetric to rule PAR$_1$.)

$$\frac{P \vdash \mathcal{G} \xrightarrow{l} P' \vdash \mathcal{G}' \quad \mathrm{bn}(l) \cap \mathrm{fn}(Q) = \emptyset}{\dfrac{P \vdash \mathcal{G} \quad Q \vdash \mathcal{H}}{P \mid Q \vdash \mathcal{G} \mid \mathcal{H}}\,\text{H-MIX} \xrightarrow{l} \dfrac{P' \vdash \mathcal{G}' \quad Q \vdash \mathcal{H}}{P' \mid Q \vdash \mathcal{G}' \mid \mathcal{H}}\,\text{H-MIX}}\,\text{PAR}_1$$

$$\frac{P \vdash \mathcal{G} \xrightarrow{l} P' \vdash \mathcal{G}' \qquad Q \vdash \mathcal{H} \xrightarrow{l'} Q' \vdash \mathcal{H}' \qquad \mathrm{bn}(l) \cap \mathrm{bn}(l') = \emptyset}{\dfrac{P \vdash \mathcal{G} \quad Q \vdash \mathcal{H}}{P \mid Q \vdash \mathcal{G} \mid \mathcal{H}}\,\text{H-MIX} \xrightarrow{(l \| l')} \dfrac{P' \vdash \mathcal{G}' \quad Q' \vdash \mathcal{H}'}{P' \mid Q' \vdash \mathcal{G}' \mid \mathcal{H}'}\,\text{H-MIX}}\,\text{SYN}$$

Rules PAR_1 and PAR_2 transform one of the two parallel components given that the transformation preserves non-interference, *i.e.*, disjointness of names. This condition follows from the requirement of distinct names in hyperenvironments, and gives the usual side-condition for rules PAR_1 and PAR_2 that one would expect for the internal π-calculus (*cf.* [Sangiorgi 1993]). Rule SYN synchronises transformations of parallel components into a single transformation labelled with the unordered pair of the respective labels—we assume that l and l' are not pairs themselves, as interactions in HCP have two parties. We write these unordered pairs of labels as $(l \parallel l')$, to evoke the parallel combination of two transformations. Formally, for all l and l', $(l \parallel l') = (l' \parallel l)$. Again, the condition on disjointness of bound names arises from the well-formedness of the resulting hyperenvironments. There are no transitions for rule H-MIX_0. Indeed, its corresponding term $\mathbf{0}$ is the terminated program.

The rule below captures the standard propagation of unrestricted actions of the π-calculus. The extracted side-condition $x *_{P'} y$ (which does not look at types) is induced by the name partitioning required for typing P' in the target.

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : A^\perp \xrightarrow{l} P' \vdash \mathcal{G}' \mid \Gamma', x : A \mid \Delta', y : A^\perp \qquad x, y \notin \mathrm{cn}(l) \qquad x *_{P'} y}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : A^\perp}{(\nu xy) P \vdash \mathcal{G} \mid \Gamma, \Delta}\,\text{H-CUT} \xrightarrow{l} \dfrac{P' \vdash \mathcal{G}' \mid \Gamma', x : A \mid \Delta', y : A^\perp}{(\nu xy) P' \vdash \mathcal{G}' \mid \Gamma', \Delta'}\,\text{H-CUT}}\,\text{RES}$$

Communication. Communication is captured by simplifying applications of rule H-CUT, given by the transformations below, one for each type of dual actions.

$$\frac{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : 1 \mid \Delta, y : \bot}{(x[] : 1 \parallel y() : \bot)}\; P' \vdash \mathcal{G}' \mid \Gamma'}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : 1 \mid \Delta, y : \bot}{(\nu xy) P \vdash \mathcal{G} \mid \Gamma, \Delta}\,\text{H-CUT}}\,1\bot \xrightarrow{\tau} P' \vdash \mathcal{G}' \mid \Gamma'$$

$$\frac{\dfrac{P \vdash \mathcal{G} \mid \Gamma, \Delta, x : A \otimes B \mid \Theta, y : A^\perp \parallel B^\perp}{(x[x'] : A \otimes B \parallel y(y') : A^\perp \parallel B^\perp)}\; P' \vdash \mathcal{G} \mid \Gamma, x : B \mid \Delta, x' : A \mid \Theta, y : B^\perp, y' : A^\perp}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, \Delta, x : A \otimes B \mid \Theta, y : A^\perp \parallel B^\perp}{(\nu xy) P \vdash \mathcal{G} \mid \Gamma, \Delta, \Theta}\,\text{H-CUT}}\,\otimes\parallel \xrightarrow{\tau} \dfrac{P' \vdash \mathcal{G} \mid \Gamma, x : B \mid \Delta, x' : A \mid \Theta, y : B^\perp, y' : A^\perp}{\dfrac{(\nu x'y') P' \vdash \mathcal{G} \mid \Gamma, x : B \mid \Delta, \Theta, y : B^\perp}{(\nu xy)(\nu x'y') P' \vdash \mathcal{G} \mid \Gamma, \Delta, \Theta}\,\text{H-CUT}}\,\text{H-CUT}$$

These transformations do not interact with the context nor have any effect on the types of the conclusions besides shuffling (*cf.* Example 3.4 and Theorem 3.9). Hence, they represent internal actions and we label them with τ, as common for process calculi.

Example 3.1. Let $P = (\nu xy)(x[x'].Q \mid y(y').z().R)$ for some Q and R such that P is well-typed. Then, we have the following transitions.

$$\begin{aligned} P &\xrightarrow{\tau} (\nu xy)(\nu x'y')(Q \mid z().R) && \text{by } \otimes\parallel, \text{SYN, and the axioms for } x[y] \text{ and } x(y) \\ &\xrightarrow{z()} (\nu xy)(\nu x'y')(Q \mid R) && \text{by RES, } \text{PAR}_2, \text{ and the axiom for } z(). \end{aligned}$$

Remark 3.2. The reader familiar with linear logic might recognise that our transition rules for communications evoke cut reductions in CLL: the way in which types are matched and deconstructed is similar. The key difference is that we do not need to permute cuts in proofs (commuting conversions) until they reach the rule applications that formed the types being deconstructed. This

is because we can *observe* what we need from our transition labels, rather than having to inspect the structure of the proofs for the premises of our transition rules.

Delayed actions. HCP supports the notion of "delayed actions", originally introduced for the π-calculus to formulate non-blocking I/O actions [Merro and Sangiorgi 2004]. Delayed actions allow actions under a prefix to be executed (observed), as long as they do not interfere with the prefix. The following rule delays an input action.

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x' : A, x : B \xrightarrow{l} P' \vdash \mathcal{G}' \mid \Gamma', x' : A, x : B \quad x, x' \notin \mathrm{cn}(l) \quad x \circledast_{P'} x'}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x' : A, x : B}{x(x').P \vdash \mathcal{G} \mid \Gamma, x : A \parr B}\parr \xrightarrow{l} \dfrac{P' \vdash \mathcal{G}' \mid \Gamma', x' : A, x : B}{x(x').P' \vdash \mathcal{G}' \mid \Gamma', x : A \parr B}\parr}\ :\!\parr$$

Any transition that does not depend on or separates the parameters of rule $\parr$ (names x and x' and their types A and B) is propagated. This condition is verified by checking in the premise that A and B are still available in the hyperenvironment after the transition. At the process level, this corresponds to checking that the names x and x' are not in the label l and that P' supports a partition that does not separate x and x'.

HCP supports also a generalised version of self-synchronisation, originally introduced by Merro and Sangiorgi [2004] together with delayed actions to model self-communication. This captures that prefixes are truly non-blocking. The idea is to execute a prefix and a non-interfering action from its continuation at the same time. This is the self-synchronisation rule for input actions.

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x' : A, x : B \xrightarrow{l} P' \vdash \mathcal{G}' \mid \Gamma', x' : A, x : B \quad x, x' \notin \mathrm{cn}(l) \quad x *_P \mathrm{fn}(l) \quad x \circledast_{P'} x'}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x' : A, x : B}{x(x').P \vdash \mathcal{G} \mid \Gamma, x : A \parr B}\parr \xrightarrow{(l \| x(x') : A \parr B)} P' \vdash \mathcal{G}' \mid \Gamma', x' : A, x : B}\ |\!\parr$$

The rule is essentially a combination of the transition axiom for the input prefix and the rule for delaying its execution.

The rules for delayed execution and self-synchronisation of the remaining prefixes are obtained likewise, below (we omit the rules for 1). In rule $:\!\bot$, the extracted premise $\mathrm{fn}(P') \neq \emptyset$ ensures, as a consequence of the proof theory, that Γ' is not empty and that P' is not a parallel composition of $\mathbf{0}$.

$$\frac{P \vdash \mathcal{G} \mid \Gamma \xrightarrow{l} P' \vdash \mathcal{G}' \mid \Gamma' \quad \mathrm{fn}(P') \neq \emptyset}{\dfrac{P \vdash \mathcal{G} \mid \Gamma}{x().P \vdash \mathcal{G} \mid \Gamma, x : \bot}\bot \xrightarrow{l} \dfrac{P' \vdash \mathcal{G}' \mid \Gamma'}{x().P' \vdash \mathcal{G}' \mid \Gamma', x : \bot}\bot}\ :\!\bot \qquad \frac{P \vdash \mathcal{G} \mid \Gamma \xrightarrow{l} P' \vdash \mathcal{G}' \quad x *_{x().P} \mathrm{fn}(l)}{\dfrac{P \vdash \mathcal{G} \mid \Gamma}{x().P \vdash \mathcal{G} \mid \Gamma, x : \bot}\bot \xrightarrow{(l \| x() : \bot)} P' \vdash \mathcal{G}'}\ |\!\bot$$

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x' : A \mid \Delta, x : B \xrightarrow{l} P' \vdash \mathcal{G}' \mid \Gamma', x' : A \mid \Delta', x : B \quad x, x' \notin \mathrm{cn}(l) \quad x *_{P'} x'}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x' : A \mid \Delta, x : B}{x[x'].P \vdash \mathcal{G} \mid \Gamma, \Delta, x : A \otimes B}\otimes \xrightarrow{l} \dfrac{P' \vdash \mathcal{G}' \mid \Gamma', x' : A \mid \Delta', x : B}{x[x'].P' \vdash \mathcal{G}' \mid \Gamma', \Delta', x : A \otimes B}\otimes}\ :\!\otimes$$

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x' : A \mid \Delta, x : B \xrightarrow{l} P' \vdash \mathcal{G}' \mid \Gamma', x' : A \mid \Delta', x : B \quad x, x' \notin \mathrm{cn}(l) \quad x *_{x[x'].P} \mathrm{fn}(l) \quad x *_{P'} x'}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x' : A \mid \Delta, x : B}{x[x'].P \vdash \mathcal{G} \mid \Gamma, \Delta, x : A \otimes B}\otimes \xrightarrow{(x[x'] : A \otimes B \| l)} P' \vdash \mathcal{G}' \mid \Gamma', x' : A \mid \Delta', x : B}\ |\!\otimes$$

Example 3.3. The lts of HCP recalls full β-reduction for the λ-calculus. Consider again the process from Example 3.1: $P = (\nu xy)(x[x'].Q \mid y(y').z().R)$ for some Q and R such that P is well-typed. Because of delayed actions, we might observe the action on z first. By rules RES, PAR$_2$ and $:\!\parr$, and

the axiom for $z()$, we get the transition $P \xrightarrow{z()} (\nu xy)(x[x'].Q \mid y(y').R) = P'$. Self-synchronisation can give rise to self-communication. Consider the self-communicating process $S = (\nu wz)\, w[].P$, where P is as above. By $1\bot$, $|\otimes$, and the transition above we get $S \xrightarrow{\tau} P'$.

Example 3.4. Consider $P \vdash v : \bot, w : 1 \mid x : 1 \mid y : \bot, z : 1$ and $P \vdash w : 1 \mid v : \bot, x : 1 \mid y : \bot, z : 1$ for $P = v().w[].x[].\mathbf{0} \mid y().z[].\mathbf{0}$. Both have a transition synchronising x and y (derived using rules $1\bot$ and SYN, and the axioms for $\bot$ and 1) leading to $P' \vdash v : \bot, w : 1 \mid z : 1$ for $P' = v().w[].\mathbf{0} \mid z[].\mathbf{0}$. Let $Q = (\nu xy)P$. $Q \vdash v : \bot, w : 1 \mid z : 1$ has a τ-transition to $P' \vdash v : \bot, w : 1 \mid z : 1$ derived using rule $1\bot$; observe that types are preserved. $Q \vdash w : 1 \mid v : \bot, z : 1$ can also perform the same synchronisation and reach $P' \vdash v : \bot, w : 1 \mid z : 1$; here types are preserved but v has been shuffled.

3.2 Additives

The derivation rules for selection ($\oplus_1$,$\oplus_2$) and choice ($\&$) are given below and are obtained with the same technique as for multiplicatives. There are left and right rules for actions, delayed actions, and communications. They are all symmetric. We omit the right cases here.

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x : A}{x \triangleleft \mathsf{inl}.P \vdash \mathcal{G} \mid \Gamma, x : A \oplus B}\,\oplus_1 \xrightarrow{x \triangleleft \mathsf{inl} : A \oplus_1 B} P \vdash \mathcal{G} \mid \Gamma, x : A$$

$$\frac{P \vdash \Gamma, x : A \quad Q \vdash \Gamma, x : B}{x \triangleright \{\mathsf{inl} : P; \mathsf{inr} : Q\} \vdash \Gamma, x : A \,\&\, B}\,\& \xrightarrow{x \triangleright \mathsf{inl} : A \&_1 B} P \vdash \Gamma, x : A$$

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x : A \oplus B \mid \Delta, y : A^\bot \,\&\, B^\bot \xrightarrow{(x \triangleleft \mathsf{inl} : A \oplus_1 B \,\|\, y \triangleright \mathsf{inl} : A^\bot \&_1 B^\bot)} P' \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : A^\bot}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A \otimes B \mid \Delta, y : A^\bot \parr B^\bot}{(\nu xy)P \vdash \mathcal{G} \mid \Gamma, \Delta}\,\text{H-CUT} \xrightarrow{\tau} \dfrac{P' \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : A^\bot}{(\nu xy)P' \vdash \mathcal{G} \mid \Gamma, \Delta}\,\text{H-CUT}}\,\oplus_1\&$$

The rules for delayed and self-synchronising selection are straightforward.

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x : A \xrightarrow{l} P' \vdash \mathcal{G}' \mid \Gamma', x : A \quad x \notin \mathrm{cn}(l)}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A}{x \triangleleft \mathsf{inl}.P \vdash \mathcal{G} \mid \Gamma, x : A \oplus B}\,\oplus_1 \xrightarrow{l} \dfrac{P' \vdash \mathcal{G}' \mid \Gamma', x : A}{x \triangleleft \mathsf{inl}.P' \vdash \mathcal{G}' \mid \Gamma', x : A \oplus B}\,\oplus_1}\,:\oplus_1$$

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x : A \xrightarrow{l} P' \vdash \mathcal{G}' \mid \Gamma, x : A \quad x \notin \mathrm{cn}(l) \quad x *_{x \triangleleft \mathsf{inl}.P} \mathrm{fn}(l)}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A}{x \triangleleft \mathsf{inl}.P \vdash \mathcal{G} \mid \Gamma, x : A \oplus B}\,\oplus_1 \xrightarrow{(x \triangleleft \mathsf{inl} : A \oplus_1 B \,\|\, l)} P' \vdash \mathcal{G}' \mid \Gamma, x : A}\,|\oplus_1$$

We choose not to define rules for delayed or self-synchronising choices, since rule $\&$ does not allow for internal independent components ($\mathcal{G}$).

Remark 3.5. If we wished to allow for delayed choices, we could add the following rule. The rule allows for delaying a choice if its two branches simultaneously undergo transformations with the same label and to targets with no parallel components. (We omit the rule for self-synchronisation.)

$$\frac{P_i \vdash \Gamma', x : A_i \xrightarrow{l} P_i' \vdash \Gamma', x : A_i \quad x \notin \mathrm{cn}(l) \quad |\mathrm{np}(P_i')| = 1 \quad \text{for } i \in \{1, 2\}}{\dfrac{P_1 \vdash \Gamma, x : A_1 \quad P_2 \vdash \Gamma, x : A_2}{x \triangleright \{\mathsf{inl} : P_1; \mathsf{inr} : P_2\} \vdash \Gamma, x : A_1 \,\&\, A_2}\,\& \xrightarrow{l} \dfrac{P' \vdash \Gamma', x : A_1 \quad Q' \vdash \Gamma', x : A_2}{x \triangleright \{\mathsf{inl} : P_1'; \mathsf{inr} : P_2'\} \vdash \Gamma, x : A_1 \,\&\, A_2}\,\&}\,:\&_0$$

Remark 3.6. Keeping the analogy with full β-reduction (Example 3.1), we could add the following transition rule for lifting internal actions by a choice branch (we omit the symmetric rule).

$$\frac{P \vdash \Gamma, x : A \xrightarrow{\tau} P' \vdash \Gamma, x : A}{\dfrac{P \vdash \Gamma, x : A \quad Q \vdash \Gamma, x : B}{x \triangleright \{\mathsf{inl} : P; \mathsf{inr} : Q\} \vdash \Gamma, x : A \,\&\, B}\,\& \xrightarrow{\tau} \dfrac{P' \vdash \Gamma, x : A \quad Q \vdash \Gamma, x : B}{x \triangleright \{\mathsf{inl} : P'; \mathsf{inr} : Q\} \vdash \Gamma, x : A \,\&\, B}\,\&}\;:\&_1$$

We choose not to, purely because it is unintuitive that a branch may perform any kind of computation before it is selected. Moreover, the rule does not change the expressiveness of HCP and its behavioural theory (our semantic equivalences abstract from internal actions, *cf.* Section 4).

3.3 Links

There are two transitions for AX and are given by the (symmetric) axioms below.

$$\frac{}{x \leftrightarrow y \vdash x : A^\perp, y : A}\,\mathrm{AX} \xrightarrow{x \leftrightarrow y : \mathrm{AX}\, A} \mathbf{0} \vdash \varnothing \qquad \frac{}{x \leftrightarrow y \vdash x : A^\perp, y : A}\,\mathrm{AX} \xrightarrow{y \leftrightarrow x : \mathrm{AX}\, A^\perp} \mathbf{0} \vdash \varnothing$$

The two transitions differ only for the order of names in the label to capture the symmetry of the link. Rule AXCUT below corresponds to the cut of rule AX.

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x : A^\perp, y : A \mid \Delta, z : A^\perp \xrightarrow{x \leftrightarrow y : \mathrm{AX}\, A} P' \vdash \mathcal{G}' \mid \Gamma', z : A^\perp}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : A^\perp, y : A \mid \Delta, z : A^\perp}{(\nu yz) P \vdash \mathcal{G} \mid \Gamma, \Delta, x : A^\perp}\,\text{H-CUT} \xrightarrow{\tau} P'\{x/z\} \vdash \mathcal{G}' \mid \Gamma', x : A^\perp}\;\text{AXCUT}$$

3.4 Exponentials

In HCP, clients can interact with servers in three ways: requesting an instance of the service provided by the server, duplicating a server, or disposing of a server.

Requesting an instance. Client requests for server instances are modelled by the following rules, for the prefixes and their communication.

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x' : A}{?x[x'].P \vdash \mathcal{G} \mid \Gamma, x : ?A}\,? \xrightarrow{?x[x'] : ?A} P \vdash \mathcal{G} \mid \Gamma, x' : A \qquad \frac{P \vdash ?\Gamma, x' : A}{!x(x').P \vdash ?\Gamma, x : !A}\,! \xrightarrow{!x(x') : !A} P \vdash ?\Gamma, x' : A$$

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

Instance requests can be delayed and self-synchronise as follows.

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x' : A \xrightarrow{l} P' \vdash \mathcal{G}' \mid \Gamma, x' : A \quad x, x' \notin \mathrm{cn}(l)}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x' : A}{?x[x'].P \vdash \mathcal{G} \mid \Gamma, x : ?A}\,? \xrightarrow{l} \dfrac{P' \vdash \mathcal{G}' \mid \Gamma, x' : A}{?x[x'].P' \vdash \mathcal{G}' \mid \Delta, x : ?A}\,?}\;:?$$

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x' : A \xrightarrow{l} P' \vdash \mathcal{G}' \mid \Gamma, x' : A \quad x, x' \notin \mathrm{cn}(l) \quad x *_{?x[x'].P} \mathrm{fn}(l)}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x' : A}{?x[x'].P \vdash \mathcal{G} \mid \Gamma, x : ?A}\,? \xrightarrow{(?x[x'] : ?A \,\|\, l)} P' \vdash \mathcal{G}' \mid \Gamma, x' : A}\;|?$$

Server duplication. Server duplication is captured by the following axioms and synchronisation rule. Since servers might contain dependencies, *i.e.*, client terms (all names of $?\Gamma$ in rule !), their duplication must be propagated to their dependencies before the new copies can be used. We write $P\sigma$, $\Gamma\sigma$, and $x\sigma$ for the application of a name substitution σ to a process, environment, and name.

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x_1 : ?A, x_2 : ?A}{?x[x_1, x_2].P \vdash \mathcal{G} \mid \Gamma, x : ?A}\,\mathrm{C} \xrightarrow{?x[x_1,x_2]:?_{\mathrm{c}}A} \frac{P \vdash \mathcal{G} \mid \Gamma, x_1 : ?A, x_2 : ?A}{x_1(x_2).P \vdash \mathcal{G} \mid \Gamma, x_1 : ?A \parr ?A}\,\parr$$

$$P_i = P\sigma_i \quad \Gamma_i = \Gamma\sigma_i \quad \mathrm{fn}(P) = \{x', z_1, \ldots, z_n\} \quad \text{for } i \in \{1, 2\}$$

$$\frac{P \vdash\, !\Gamma, x' : A}{!x(x').P \vdash\, ?\Gamma, x : !A}\,!$$

$$\downarrow\, !x(x_1, x_2) : !_{\mathrm{c}}A$$

$$\frac{\dfrac{\dfrac{P_1 \vdash\, ?\Gamma_1, x'\sigma_1 : A}{!x_1(x'\sigma_1).P_1 \vdash\, ?\Gamma_1, x_1 : !A}\,! \quad \dfrac{P_2 \vdash\, ?\Gamma_2, x'\sigma_2 : A}{!x_2(x'\sigma_2).P_2 \vdash\, ?\Gamma_2, x_2 : !A}\,!}{!x_1(x'\sigma_1).P_1 \mid !x_2(x'\sigma_2).P_2 \vdash\, ?\Gamma_1, x_1 : !A \mid ?\Gamma_2, x_2 : !A}\,\text{H-MIX}}{x_1[x_2].(!x_1(x'\sigma_1).P \mid !x_2(x'\sigma_2).P) \vdash\, ?\Gamma_1, ?\Gamma_2, x_1 : !A \otimes !A}\,\otimes}{?z_1[z_1\sigma_1, z_1\sigma_2]. \ldots ?z_n[z_n\sigma_1, z_n\sigma_2].x_1[x_2].(!x_1(x'\sigma_1).P_1 \mid !x_2(x'\sigma_2).P_2) \vdash\, ?\Gamma, x_1 : !A \otimes !A}\,\mathrm{C}$$

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x : ?A \mid ?\Delta, y : !A^{\perp} \xrightarrow{(?x[x_1,x_2]:?_{\mathrm{c}}A \| !y(y_1,y_2):!_{\mathrm{c}}A^{\perp})} P' \vdash \mathcal{G} \mid \Gamma, x_1 : ?A \parr ?A \mid ?\Delta, y_1 : !A^{\perp} \otimes !A^{\perp}}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : ?A \mid ?\Delta, y : !A^{\perp}}{(\nu xy)P \vdash \mathcal{G} \mid \Gamma, ?\Delta}\,\text{H-CUT} \xrightarrow{\tau} \dfrac{P' \vdash \mathcal{G} \mid \Gamma, x_1 : ?A \parr ?A \mid ?\Delta, y_1 : !A^{\perp} \otimes !A^{\perp}}{(\nu x_1 y_1)P' \vdash \mathcal{G} \mid \Gamma, ?\Delta}\,\text{H-CUT}}\,!\mathrm{c}$$

Server duplication is done slightly differently compared to HCP's predecessor, Classical Processes (CP) [Wadler 2014]. Because in CP rules c and w lack proof terms (they can be applied "silently"), the duplication of all the resources that a server needs to introduce a new copy is not visible at the process level, whereas it is in HCP. Our semantics has thus a clearer computational interpretation. Also, the duplication of a server in CP is handled by the synchronisation rule for cut of rule ! with rule c ([Wadler 2014, Fig. 3]), whereas our semantics respects locality: the new servers appears exactly where the original server was, and our transition rule is independent of the presence of any cut in the context.

Another difference with CP's server duplication is the exchange following a duplication request (terms $y_1(y_2)$ and $y_1[y_2]$ in the targets of the transitions). This exchange is suggested by the typing: server copies must be in parallel components (they might be used concurrently) whereas rule c assumes copies are in the same environment. We could do away with these exchanges by adding *external contraction* to our proof theory, a variant of rule c where the two names to appear in separate environments [Avron 1991]. We chose not to, for simplicity of our proof theory. Because the exchange can only take place after the server sends all requests to duplicate its dependencies and because server copies can only be used after the exchange has been done, it has a clear operational interpretation: it is an acknowledgement.

Duplication requests can be delayed and self-synchronise.

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x_1 : ?A, x_2 : ?A \xrightarrow{l} P' \vdash \mathcal{G}' \mid \Gamma', x_1 : ?A, x_2 : ?A \quad x, x_1, x_2 \notin \mathrm{cn}(l) \quad x_1 \circledast_{P'} x_2}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x_1 : ?A, x_2 : ?A}{?x[x_1, x_2].P \vdash \mathcal{G} \mid \Gamma, x : ?A}\,\mathrm{C} \xrightarrow{l} \dfrac{P' \vdash \mathcal{G}' \mid \Gamma', x_1 : ?A, x_2 : ?A}{?x[x_1, x_2].P' \vdash \mathcal{G}' \mid \Gamma', x : ?A}\,\mathrm{C}}\,:\mathrm{c}$$

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x_1 : ?A, x_2 : ?A \xrightarrow{l} P' \vdash \mathcal{G}' \mid \Gamma', x_1 : ?A, x_2 : ?A \quad x, x_1, x_2 \notin \mathrm{cn}(l) \quad x_1 \circledast_{P'} x_2 \quad x *_{?x[x_1,x_2].P} \mathrm{fn}(l)}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x_1 : ?A, x_2 : ?A}{?x[x_1,x_2].P \vdash \mathcal{G} \mid \Gamma, x : ?A}\,\mathrm{C} \xrightarrow{(?x[x_1,x_2] : ?_{\mathrm{c}}A \| l)} \dfrac{P' \vdash \mathcal{G}' \mid \Gamma', x_1 : ?A, x_2 : ?A}{x_1(x_2).P' \vdash \mathcal{G}' \mid \Gamma', x_1 : ?A \parr ?A}\,\parr}\,|\mathrm{c}$$

Duplication requests by a server being duplicated cannot be delayed after the operation acknowledgement: $y_1[y_2]$ will always be the last operation before the new instances become available.

Example 3.7. Continuing Example 2.3, we can formally observe that using different clients that send different bits yields different results.

$$(\nu xy)\left(Client^{01}_{xz} \mid Server_y\right) \xrightarrow{\tau} \cdots \xrightarrow{\tau} \xrightarrow{z \triangleleft \mathsf{inl}} \xrightarrow{z[]} \mathbf{0} \mid \mathbf{0}$$
$$(\nu xy)\left(Client^{11}_{xz} \mid Server_y\right) \xrightarrow{\tau} \cdots \xrightarrow{\tau} \xrightarrow{z \triangleleft \mathsf{inr}} \xrightarrow{z[]} \mathbf{0} \mid \mathbf{0}$$

We can use the service multiple times by duplicating it.

$$(\nu xy)\left(?x[x_1,x_2].\left(Client^{01}_{x_1z_1} \mid Client^{11}_{x_2z_2}\right) \mid Server_y\right) \xrightarrow{\tau} \cdots \xrightarrow{\tau} \xrightarrow{z_1 \triangleleft \mathsf{inl}} \xrightarrow{z_2 \triangleleft \mathsf{inr}} \xrightarrow{z_1[]} \xrightarrow{z_2[]} \mathbf{0} \mid \mathbf{0} \mid \mathbf{0} \mid \mathbf{0}$$

Remark 3.8. In connection to Remark 3.6, we could add the following transition rule for lifting internal actions by servers.

$$\frac{P \vdash ?\Gamma, x' : A \xrightarrow{\tau} P' \vdash ?\Gamma', x' : A}{\dfrac{P \vdash ?\Gamma, x' : A}{!x(x').P \vdash ?\Gamma, x : !A}\,! \xrightarrow{\tau} \dfrac{P \vdash ?\Gamma', x' : A}{!x(x').P' \vdash ?\Gamma', x : !A}\,!}\,:!$$

We choose to exclude it because it might be surprising that a server performs any kind of computation before it is triggered. However, in general, the rule might be interesting to allow servers to "optimise" themselves before being run.

Server disposal. Server disposal is captured by the following rules, again for the dual prefixes and their communication. Since servers might contain dependencies, their disposal must be propagated before the operation is acknowledged to the client.

$$\frac{P \vdash \mathcal{G} \mid \Gamma}{?x[].P \vdash \mathcal{G} \mid \Gamma, x : ?A}\,\mathrm{W} \qquad \frac{\mathrm{fn}(P) = \{x', z_1, \ldots, z_n\}}{\dfrac{P \vdash ?\Gamma, x' : A}{!x(x').P \vdash ?\Gamma, x : !A}\,! \xrightarrow{!x() : !_{\mathrm{c}}A} \dfrac{\dfrac{\dfrac{}{\mathbf{0} \vdash \varnothing}\,\mathrm{H\text{-}MIX}_0}{x[].\mathbf{0} \vdash x : 1}\,1}{?z_1[]. \ldots ?z_n[].x[].\mathbf{0} \vdash ?\Gamma, x : 1}\,\mathrm{W}}$$

$$\downarrow ?x[] : ?_{\mathrm{c}}A \qquad\qquad \frac{}{x().P \vdash \mathcal{G} \mid \Gamma, x : \bot}\,\bot$$

$$\frac{P \vdash \mathcal{G} \mid \Gamma, x : ?A \mid ?\Delta, y : !A^{\perp} \xrightarrow{(?x[] : ?_{\mathrm{c}}A \| !y() : !_{\mathrm{c}}A^{\perp})} P \vdash \mathcal{G} \mid \Gamma, x : \bot \mid !\Delta, y : 1}{\dfrac{P \vdash \mathcal{G} \mid \Gamma, x : ?A \mid ?\Delta, y : !A^{\perp}}{(\nu xy)P \vdash \mathcal{G} \mid \Gamma, ?\Delta}\,\mathrm{H\text{-}CUT} \xrightarrow{\tau} \dfrac{P \vdash \mathcal{G} \mid \Gamma, x : \bot \mid ?\Delta, y : 1}{(\nu xy)P' \vdash \mathcal{G} \mid \Gamma, ?\Delta}\,\mathrm{H\text{-}CUT}}\,!\mathrm{w}$$

As for instance requests, disposal requests can be delayed and self-synchronise (we omit the proof transformations). Akin to duplication, disposal requests by a server being disposed cannot be delayed after the operation acknowledgement: $x()$ will always be the last operation.

3.5 Subject Reduction and Progress

All transition rules are derived from proof transformations that preserve provability. Rules for τ-transitions preserve also types: the types in (the judgement in) the conclusion remain unchanged. Thus, we immediately obtain the theorem below. Given a proof transition label l, we write $\lfloor l \rfloor$ for the same label where the logical part (the blue part) is stripped, *e.g.* $\lfloor x[] : 1 \rfloor = x[]$.

Theorem 3.9 (Subject reduction). *Let P and Q be well-typed. Then:*

- *If* $\boxed{P \vdash \mathcal{G}} \xrightarrow{l} \boxed{Q \vdash \mathcal{H}}$ *then,* $P \xrightarrow{\lfloor l \rfloor} Q$ *and* $\mathcal{G} \sqcup\!\sqcup \mathcal{H}$ *iff* $l = \tau$.
- *If* $P \vdash \mathcal{G}$ *and* $P \xrightarrow{l} Q$, *then* $\boxed{P \vdash \mathcal{G}} \xrightarrow{l'} \boxed{Q \vdash \mathcal{H}}$ *for some* $\mathcal{H}$ *and* l' *such that* $l = \lfloor l' \rfloor$.

Theorem 3.9 formalises that τ-transitions of a process P have no dependencies on the context, since they do not influence the types of P. This matches the intuition of lts semantics for the π-calculus, where τ-transitions capture internal "unobservable" moves. In HCP, performing unobservable moves coincides with type-preserving proof transitions.

The next results use "saturated" transitions (Milner's double arrow). Formally, $\Longrightarrow$ is the smallest relation such that: $P \overset{\tau}{\Longrightarrow} P$ for all P; and if $P \overset{\tau}{\Longrightarrow} P'$, $P' \xrightarrow{l} Q'$ and $Q' \overset{\tau}{\Longrightarrow} Q$, then $P \overset{l}{\Longrightarrow} Q$.

We write l_x to range over labels that denote actions on a name x: $x[]$, $x()$, $x[y]$, $x(y)$, *etc.* Actions typed by separate environments can always be observed in parallel.

Lemma 3.10. *If* $P \vdash \mathcal{G} \mid \Gamma, x : A \mid \Delta, y : B$, $P \overset{l_x}{\Longrightarrow} P'$, *and* $P \overset{l_y}{\Longrightarrow} P''$, *then there exists* Q *such that* $P \overset{(l_x \| l_y)}{\Longrightarrow} Q$ *(up to* α*-renaming).*

Well-typed processes enjoy a notion of readiness on their free channels, captured by the separation of environments. Assume that $P \vdash \mathcal{G} \mid \Gamma$. Then, there is always a name x in $\mathrm{cn}(\Gamma)$ such that $P \overset{l_x}{\Longrightarrow} P'$. In other words, every environment in the hyperenvironment used to type a process contains at least one name where the process can perform an action (possibly after τ-transitions).

Theorem 3.11 (Readiness). *Let* $P \vdash \Gamma_1 \mid \cdots \mid \Gamma_n$. *For every* $i \in [1, n]$, *there exist* $x \in \mathrm{cn}(\Gamma_i)$, l_x, *and* P' *such that* $P \overset{l_x}{\Longrightarrow} P'$.

As a corollary of Theorem 3.11, we get that well-typed processes that are not terminated can always progress, because the hyperenvironment used to type a process $P \not\equiv \mathbf{0}$ can never be empty.

Corollary 3.12 (Progress). *If* P *is well-typed and* $P \not\equiv \mathbf{0}$, *then* $P \xrightarrow{l} P'$ *for some* l *and* P'.

4 BEHAVIOURAL THEORY

We study the behavioural theory of HCP under the lenses of two classical notions of behavioural equivalence: bisimulation and barbed congruence. The first has emerged as a powerful operational method for proving equivalence of programs in various kinds of languages, due to the associated coinductive proof method. The second is the equivalent of contextual equivalence in concurrency.

Bisimilarity. The standard notion of strong bisimilarity can be instantiated on the lts of HCP.

Definition 4.1 (Strong bisimilarity). A symmetric relation $\mathcal{R}$ on processes is a strong bisimulation if $P \mathcal{R} Q$ implies that if $P \xrightarrow{l} P'$ then $Q \xrightarrow{l} Q'$ for some Q' such that $P' \mathcal{R} Q'$. Strong bisimilarity is the largest relation $\sim$ that is a strong bisimulation.

Strong bisimilarity can discriminate processes whose behaviours differs only by τ-transitions. For instance, $(\nu xy)(x[].\mathbf{0} \mid y().z[].\mathbf{0})$ and $z[].\mathbf{0}$ are not bisimilar exclusively because the first has a

τ-transition. Instead, no HCP process composed in parallel with either of them would be able to tell the difference: τ-transitions have no effect on or interaction with the context. This deficiency of strong bisimilarity is well-known and has been widely studied since Milner's seminal works on CCS [Milner 1989]. The solution is to define bisimilarity on the saturated transition relation $\Longrightarrow$.

Definition 4.2 (Bisimilarity). A symmetric relation $\mathcal{R}$ on processes is a bisimulation if $P \mathcal{R} Q$ implies that if $P \overset{l}{\Longrightarrow} P'$ then $Q \overset{l}{\Longrightarrow} Q'$ for some Q' such that $P' \mathcal{R} Q'$. Bisimilarity is the largest relation $\approx$ that is a bisimulation.

It follows from the inclusion of rule $=_\alpha$ in the specification of the lts of HCP that $=_\alpha \subsetneq \sim$. Also, from the definition of saturation it follows that $\sim \subsetneq \approx$, as usual for saturation-based behavioural equivalences [Brengos et al. 2015].

Fact 4.3. *Assume processes are well-typed. As in standard process algebras, parallel composition and* $\mathbf{0}$ *obey the laws of an abelian monoid under (strong) bisimilarity. Formally, for any* P, Q, *and* R:

$$P \sim P \mid \mathbf{0} \qquad P \mid Q \sim Q \mid P \qquad P \mid (Q \mid R) \sim (P \mid Q) \mid R$$

Restriction distributes over actions, parallel composition, and restriction, provided that they do not depend on the restricted channel. For $x, y \notin (\mathrm{fn}(R) \cup \mathrm{cn}(\pi) \cup \{x', y'\})$:

$$(\nu xy)(P \mid R) \sim (\nu xy)(P) \mid R \quad (\nu xy)\pi.P \sim \pi.(\nu xy)P \qquad (\nu xy)(\nu x'y')P \sim (\nu x'y')(\nu xy)P$$

Closing a channel is non-blocking: $x[].P \sim x[].\mathbf{0} \mid P$. *Links are symmetric:* $x{\leftrightarrow}y \sim y{\leftrightarrow}x$.

Bisimilarity and strong bisimilarity are congruences, in the sense that they are preserved by all syntactic operators of HCP.

Theorem 4.4 (Congruence). *For* $\asymp \in \{\sim, \approx\}$, *if* $P \asymp Q$, *then*
(1) $P \mid R \asymp Q \mid R$ *for any* R *(and the symmetric);*
(2) $\pi.P \asymp \pi.Q$ *for any prefix* π*;*
(3) $x \triangleright \{\mathsf{inl} : R; \mathsf{inr} : P\} \asymp x \triangleright \{\mathsf{inl} : R; \mathsf{inr} : Q\}$ *for any* x *and* R *(and the symmetric);*
(4) $(\nu xy)P \asymp (\nu xy)Q$ *for any* x, y.

Bisimilarity implies type equivalence on well-typed processes.

Proposition 4.5. *If* P *and* Q *are well-typed and* $P \approx Q$, *then* $P \vdash \mathcal{G}$ *iff* $Q \vdash \mathcal{G}$.

Barbed congruence. Contextual equivalence defines as equivalent all programs that "behave in the same manner" in any given "context" [Morris 1968]. In typed languages, a context is a typed program C with a typed "hole" $\Box$ where we can plug a program P and obtain the program $C[P]$. In HCP this instantiates to the following.

Definition 4.6 (Typed context). A context C is a *$(\mathcal{G}/\mathcal{H})$-context* if $C \vdash \mathcal{H}$ is valid when the hole $\Box$ of C is considered as a process and the following rule is added to the theory of HCP:

$$\frac{}{\Box \vdash \mathcal{G}}$$

In concurrency, behaviours are characterised in terms of the observable interactions between a process and its execution environment. We start by borrowing the notion of observability used by the π-calculus.

Definition 4.7 (Observability predicates). Given a name x, the *observability predicate* $P{\Downarrow_x}$ holds iff $P \overset{l}{\Longrightarrow} Q$ for some Q and l such that $x \in \mathrm{fn}(l)$.

Example 4.8. Consider $Q_1 = x(x').y(y').P$ and $Q_2 = y(y').x(x').P$ from the Introduction. In HCP they exhibit exactly the same observations: $Q_1{\Downarrow_x}$, $Q_1{\Downarrow_y}$, $Q_2{\Downarrow_x}$, and $Q_2{\Downarrow_y}$.

Atkey [2017] argues that a suitable notion of observational equivalence must discriminate clients that use non-linear resources in the environment in different quantities. This requires some care in the definition of barbed congruence for HCP. We illustrate the reason below.

Example 4.9. Consider type equivalent processes that consume a different number of instances of the same server: $P_0 = ?x[].Q_0$, $P_1 = ?x[x'].Q_1$, $P_2 = ?x[x_1, x_2].?x_1[x'].?x_2[x''].Q_2$, and $P_3 = ?x[x_1, x_2].?x_1[].?x_2[x''].Q_3$. The observability predicate does not distinguish these processes, since $\Downarrow_x$ holds for all of them. Thus, distinguishing them requires using a process context. Due to typing, this context can only connect x to a server, and the only way that a server has to relay information to an external observer is via client requests. Thus, we end up in the same problem we started from. Consider the context $(\nu xy)(\Box \mid !y(y').?u[].?v[].?z[w].y' \leftrightarrow w)$. For any $i \in \{0, 1, 2\}$, $C[P_i]\Downarrow_u$ and $C[P_i]\Downarrow_v$. (For the case of P_0, recall that disposing of a server triggers the disposal of its dependencies.)

This kind of deficiency of the standard observability predicate is not new for session-typed calculi: Yoshida et al. [2007] faced it for selections and choices. Likewise, we introduce predicates for observing how a program may use or dispose of a non-linear resource from the context.

Definition 4.10 (Non-linear observability predicates). For x a name and t a binary tree with leaves in $\{\diamond, \dagger\}$ (for use and dispose, respectively), the *non-linear observability predicate* $\Downarrow^t_x$ (read "x is used as in t") holds on P whenever any of the following holds:

- $t = \langle\dagger\rangle$, $P \xrightarrow{l} Q$, $?x[]$ occurs in l, and $x \notin \mathrm{fn}(P)$;
- $t = \langle\diamond\rangle$, $P \xrightarrow{l} Q$, $?x[x']$ occurs in l for some x', and $x \notin \mathrm{fn}(P)$;
- $t = \langle t_1, t_2\rangle$, $P \xrightarrow{l} Q$, $?x[x_1, x_2]$ occurs in l for some x_1 and x_2, $Q\Downarrow^{t_1}_{x_1}$ and $Q\Downarrow^{t_2}_{x_2}$;
- $P \xrightarrow{l} Q$, $?x[]$, $?x[x']$, and $?x[x_1, x_2]$ do not occur in l, and $Q\Downarrow^t_x$.

Example 4.11. Consider processes P_0, P_1, and P_2 from Example 4.9. Then, $P_0\Downarrow^t_x$ iff $t = \langle\dagger\rangle$, $P_1\Downarrow^t_x$ iff $t = \langle\diamond\rangle$, $P_2\Downarrow^t_x$ iff $t = \langle\langle\diamond\rangle, \langle\diamond\rangle\rangle$, $P_3\Downarrow^t_x$ iff $t = \langle\langle\dagger\rangle, \langle\diamond\rangle\rangle$.

When Yoshida et al. [2007] faced a similar problem for selections, they extended observability to observe the payload of selections (left or right): the new predicates $\Downarrow^{\mathsf{inl}}_x$ and $\Downarrow^{\mathsf{inr}}_x$ hold on P whenever P sends on x a left or a right selection, respectively. In HCP, we do not need this extension because we can subsume these predicates using our non-linear observability predicates and appropriate contexts, as we exemplify below.

Example 4.12. Consider a pair of processes that differ exclusively for the selection they make, $P_l = x \triangleleft \mathsf{inl}.Q$ and $P_r = x \triangleleft \mathsf{inr}.Q$. There is no direct observation that distinguishes them: for any w and t, $P_l\Downarrow_w$ iff $P_r\Downarrow_w$ and $P_l\Downarrow^t_w$ iff $P_r\Downarrow^t_w$. However, we can make an indirect observation using any context that offers a choice between two observationally distinct branches: for $C = (\nu xy)(\Box \mid y \triangleright \{\mathsf{inl} : ?w[].y \leftrightarrow z; \mathsf{inr} : ?w[w'].w'().y \leftrightarrow z\})$, $C[P_0]\Downarrow^t_w$ iff $t = \langle\dagger\rangle$ and $C[P_1]\Downarrow^t_w$ iff $t = \langle\diamond\rangle$.

Definition 4.13 (Barbed congruence). Barbed congruence is the largest symmetric relation $\cong$ on well-typed HCP processes that is

- typed ($P \cong Q$ implies $P \vdash \mathcal{G}$ iff $Q \vdash \mathcal{G}$);
- context-closed (*i.e.* if $P \cong Q$ then $C[P] \cong C[Q]$ for any typed context);
- barb preserving (*i.e.* if $P \cong Q$ and $P\Downarrow_x$ then $Q\Downarrow_x$; if $P \cong Q$ and $P\Downarrow^t_x$ then $Q\Downarrow^t_x$).

On well-typed processes, bisimilarity implies typed barbed congruence.

Theorem 4.14. $\approx \subseteq \cong$.

We anticipate that also the converse of Theorem 4.14 holds but we do not prove it directly. Instead, we show in Section 6 that this holds by comparison with the denotational semantics.

5 DENOTATIONAL SEMANTICS

Inspired by the relational semantics for proofs in linear logic by Barr [1996], Atkey [2017] developed a denotational semantics for CP that interprets well-typed processes as sets of possible observable interactions on each of their free channels. In this section, we show that a similar denotational semantics can be also developed for HCP.

The main novelty, besides extending denotations to hyperenvironments, is that we rediscover denotations from a very different angle, based on HCP's notion of observable actions. Specifically, inspired by Brzozowski derivatives of regular expressions [Brzozowski 1964], we introduce "derivatives of denotations" w.r.t. HCP actions. As the derivative of a set of strings w.r.t. a character is the set of strings that can can follow that character, the derivative of a denotation w.r.t. an action is the denotation that can follow that action. This approach allows us to abstract from the syntax of HCP and study actions and interactions solely in terms of denotations. For instance, we characterise non-interference, usually a topic of operational semantics, as Fubini's equality for double antiderivatives [Fubini 1907].

Atkey's denotations do not distinguish outputs from inputs. For instance, $x \triangleleft \mathsf{inl}.x[].\mathbf{0} \vdash 1 \oplus 1$ and $x \triangleright \{\mathsf{inl} : x[].\mathbf{0}; \mathsf{inr} : x[].\mathbf{0}\} \vdash 1 \mathbin{\&} 1$ are given the same denotation. To characterise observational equivalence, Atkey resorts to the intersection of type equivalence with denotational equivalence: two processes are observationally equivalent if they have the same types and denotations. Differently, our denotations distinguish inputs from outputs, and our notion of denotational equivalence does not require type equivalence.

Interpretation of types. We define the domain of denotations for a name by recursion on the structure of its type. We tag denotations with natural numbers and use these identifiers in certain denotations to describe dependencies with other channels:

- in the denotations of $\otimes$, we use identifiers to specify which observations must be in which of two parallel components merged by a tensor;
- in the denotations of $\&$ and !, we use identifiers to specify which observations are blocked until a choice is performed or a server is used, respectively—which are the only constructs in HCP without delayed actions.

We use $\triangleleft$ and $\triangleright$ to distinguish between outputs and inputs. We write $\wp_f$ for the finite powerset.

We interpret multiplicative units as singletons because they are the types that characterise the empty output and input, respectively.

$$\{\!|1|\!\} = \{\triangleleft\} \times \mathbb{N} \times \{*\} \qquad \{\!|\bot|\!\} = \{\triangleright\} \times \mathbb{N} \times \{*\}$$

We interpret multiplicatives as cartesian products pairing denotations of their components.

$$\{\!|A \otimes B|\!\} = \{\triangleleft\} \times \mathbb{N} \times (\wp_f\mathbb{N} \times \{\!|A|\!\} \times \wp_f\mathbb{N} \times \{\!|B|\!\}) \quad \{\!|A \parr B|\!\} = \{\triangleright\} \times \mathbb{N} \times (\{\!|A|\!\} \times \{\!|B|\!\})$$

We interpret additives as coproducts and represent them as dependent pairs indexed over $\{\mathsf{inl}, \mathsf{inr}\}$.

$$\{\!|A \oplus B|\!\} = \{\triangleleft\} \times \mathbb{N} \times (\{\!|A|\!\} + \{\!|B|\!\}) \qquad \{\!|A \mathbin{\&} B|\!\} = \{\triangleright\} \times \mathbb{N} \times \wp_f\mathbb{N} \times (\{\!|A|\!\} + \{\!|B|\!\})$$

We interpret exponentials as sets of binary trees of denotations (written $\mathcal{T}$) since they characterise both unbounded (but finite) interactions of a given type and resource allocation (duplication via branchings and disposal via leaves).

$$\{\!|?A|\!\} = \{\triangleleft\} \times \mathbb{N} \times (\mathcal{T}(\{\!|A|\!\} + \{\dagger\})) \qquad \{\!|!A|\!\} = \{\triangleright\} \times \mathbb{N} \times \wp_f\mathbb{N} \times (\mathcal{T}(\{\!|A|\!\} + \{\dagger\}))$$

If we ignore identifiers in denotations, we can turn $a \in \{\!|A|\!\}$ into an element of $\{\!|A^\perp|\!\}$ by simultaneously replacing $\triangleright$ with $\triangleleft$ and *vice versa*. We extend the notion of duality from types to their interpretation and write $a \leftrightarrow b$ whenever $a \in \{\!|A|\!\}$, $b \in \{\!|A^\perp|\!\}$, and reversing the direction of triangles in a yields b up to differences in identifiers—sometimes, we abuse the terminology and call a and b dual.

Example 5.1. Consider $(\triangleleft, n, *) \in \{\!|1|\!\}$, $(\triangleright, n', *) \in \{\!|\bot|\!\}$ and write a and b for them, respectively. We have that $a \leftrightarrow b$, for any $n, n' \in \mathbb{N}$. Also, $(\triangleleft, m, \mathsf{inl}, a) \leftrightarrow (\triangleright, m', M, \mathsf{inl}, b)$ for any $m, m' \in \mathbb{N}$ and $M \in \wp_f \mathbb{N}$.

We sometimes use the wildcard _ when pattern matching denotations, *e.g.*, we write $(\triangleright, \_, *) = a$ instead of $(\triangleright, n, *) = a$ if binding n is superfluous.

We define the denotation domain of an environment as the set of all assignments that take each name to observations of its associated type. We write $\langle x_1 \mapsto a_1, \ldots, x_n \mapsto a_n \rangle$ for the assignment taking each name x_i of type A_i to its denotation $a_i \in \{\!|A_i|\!\}$. As usual, we work up to exchange and hence we interpret "," as the cartesian product and assignments as tuples.

$$\{\!|\bullet|\!\} = \{\langle\rangle\} \qquad\qquad \{\!|\Gamma, x : A|\!\} = \{\!|\Gamma|\!\} \times \{\langle x \mapsto a \rangle \mid a \in \{\!|A|\!\}\}$$

Likewise, we interpret hyperenvironments as collections of assignments from the domains of their environments and map "|" to the cartesian product. We use γ, δ to range over elements of $\{\!|\Gamma|\!\}$.

$$\{\!|\varnothing|\!\} = \{\langle\rangle\} \qquad\qquad \{\!|\mathcal{G} \mid \Gamma|\!\} = \{\!|\mathcal{G}|\!\} \times \{\langle\gamma\rangle \mid \gamma \in \{\!|\Gamma|\!\}\}$$

We write $\mathrm{id}(a)$ for the outermost identifier in a and $\mathrm{ids}(a)$ for the set of all identifiers in a (excluding dependencies used in the interpretations of $\otimes$, $\&$, and !), and extend these notations to the interpretations of environments and hyperenvironments ($\mathrm{id}(\gamma)$ is the set of all outermost identifiers in γ and $\mathrm{ids}(\gamma)$ is the set of all identifiers in γ).

Since we are using numeric identifiers to encode dependencies, it is natural to assume that identifiers are unique within a denotation and that dependencies in the interpretations of $\otimes$, $\&$, and ! are limited to the same environment and have no cycles. We refer to denotations that respect this simple identifier discipline as *well-formed*.

- We call $a \in \{\!|A|\!\}$ well-formed if its identifiers are unique and decrease along its structure (*e.g.*, if $a = (\triangleleft, n, \mathsf{inl}, a')$, then $n > \mathrm{ids}(a')$).
- We call $\gamma = \langle x_1 \mapsto a_1, \ldots, x_k \mapsto a_k \rangle \in \{\!|x_1 : A_1, \ldots, x_k : A_k|\!\}$ well-formed if its identifiers are unique and for every $i \in \{1, \ldots, k\}$:
  (1) a_i is well-formed;
  (2) if $A_i = B_1 \otimes B_2$, then $N_1, N_2 \subseteq \mathrm{ids}(\gamma)$ and $n > N_1, N_2$, where $a_i = (\triangleleft, n, N_1, \_, N_2, \_)$;
  (3) if $A_i = B_1 \,\&\, B_2$ or $A_i = !B$, then $N \subseteq \mathrm{ids}(\gamma)$ and $n > N$, where $a_i = (\triangleleft, n, N, \_, \_)$ or $a_i = (\triangleleft, n, N, \_)$.
- We call $g = \langle \gamma_1, \ldots, \gamma_k \rangle$ well-formed if $\gamma_1, \ldots, \gamma_k$ are well-formed and its identifiers are unique.

From now on, unless otherwise specified, we assume that denotations are well-formed.

Derivatives and Antiderivatives. Recall from [Brzozowski 1964] that the derivative w.r.t. a character l of a set of strings D is the set $\{a \mid la \in D\}$ of strings a that can follow l to yield a string la in D. In particular, if we consider the language accepted by a state p of a deterministic automaton and take its derivative w.r.t. an input symbol l, then we obtain the accepted language for the state reached by the automaton if we input l while in p. In light of this correspondence, Brzozowski derivatives define an automaton on accepted languages, not any automaton, but the one that is final in the coalgebraic sense thus marrying denotational and observational equivalence of automata [Bonchi et al. 2014].

Inspired by Brzozowski derivatives, we define the derivative of a set of denotations D w.r.t. to an action l as the set of the denotations that can follow l. For instance, derivation w.r.t. $x[]$ takes $\langle\langle x \mapsto (\triangleleft, n, *)\rangle\rangle \in \{\!|x : 1|\!\}$ to $\langle\rangle \in \{\!|\varnothing|\!\}$ since there are no actions left; derivation w.r.t. $x[y]$ takes $\langle\langle x \mapsto (\triangleleft, n, \emptyset, a, \emptyset, b)\rangle\rangle \in \{\!|x : A \otimes B|\!\}$ to $\langle\langle x \mapsto b\rangle, \langle y \mapsto a\rangle\rangle \in \{\!|x : B \mid y : A|\!\}$ since after y is output on x the two have separate behaviours. Below, we write $a\backslash n$ for the denotation where n is removed from all interpretations of $\otimes$ in a, *e.g.*, $(\triangleright, m, \{n\}, a)\backslash n = (\triangleright, m, \{n\}, a\backslash n)$ and

$(\triangleleft, m, \{n\}, a, \emptyset, a')\backslash n = (\triangleleft, m, \emptyset, a\backslash n, \emptyset, a'\backslash n)$); likewise for environments. We write $\cdot$ for juxtaposition, *e.g.*, $\langle\langle x \mapsto a\rangle\rangle \cdot \langle\langle y \mapsto a', z \mapsto a''\rangle\rangle$ is $\langle\langle x \mapsto a\rangle, \langle y \mapsto a', z \mapsto a''\rangle\rangle$ from $\{\!|x : A \mid y : A', z : A''|\!\}$.

Definition 5.2 (Derivatives). For D a set of denotations, the derivative $\mathrm{d}_l D$ of D w.r.t. an action l is the set defined by:

$$
\begin{aligned}
\mathrm{d}_{x[]}D &= \{g \mid g \cdot \langle\langle x \mapsto (\triangleleft, \_, *)\rangle\rangle \in D\} \\
\mathrm{d}_{x()}D &= \{g \cdot \langle\gamma\backslash n\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleright, n, *)\rangle\rangle \in D\} \\
\mathrm{d}_{x[y]}D &= \left\{ g \cdot \langle\gamma\backslash n, N' \cdot \langle x \mapsto b\rangle, \delta\backslash n, N \cdot \langle y \mapsto a\rangle\rangle \;\middle|\; \begin{array}{l} g \cdot \langle\gamma \cdot \delta \cdot \langle x \mapsto (\triangleleft, n, N, a, N', b)\rangle\rangle \in D, \\ N \subseteq \mathrm{ids}(\gamma), N' \subseteq \mathrm{ids}(\delta) \end{array} \right\} \\
\mathrm{d}_{x(y)}D &= \{g \cdot \langle\gamma\backslash n \cdot \langle x \mapsto b, y \mapsto a\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleright, n, a, b)\rangle\rangle \in D\} \\
\mathrm{d}_{x \triangleleft \mathsf{inl}}D &= \{g \cdot \langle\gamma\backslash n \cdot \langle x \mapsto a\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleleft, n, \mathsf{inl}, a)\rangle\rangle \in D\} \\
\mathrm{d}_{x \triangleright \mathsf{inl}}D &= \{g \cdot \langle\gamma\backslash n \cdot \langle x \mapsto a\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleright, n, \_, \mathsf{inl}, a)\rangle\rangle \in D\} \\
\mathrm{d}_{x \triangleleft \mathsf{inr}}D &= \{g \cdot \langle\gamma\backslash n \cdot \langle x \mapsto b\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleleft, n, \mathsf{inr}, b)\rangle\rangle \in D\} \\
\mathrm{d}_{x \triangleright \mathsf{inr}}D &= \{g \cdot \langle\gamma\backslash n \cdot \langle x \mapsto b\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleright, n, \_, \mathsf{inr}, b)\rangle\rangle \in D\} \\
\mathrm{d}_{?x[y]}D &= \{g \cdot \langle\gamma\backslash n \cdot \langle x \mapsto a\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleleft, n, \langle a\rangle)\rangle\rangle \in D, a \neq \dagger\} \\
\mathrm{d}_{!x(y)}D &= \{g \cdot \langle\gamma\backslash n \cdot \langle x \mapsto a\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleright, n, \_, \langle a\rangle)\rangle\rangle \in D, a \neq \dagger\} \\
\mathrm{d}_{?x[]}D &= \{g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleright, n, *)\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleleft, n, \langle\dagger\rangle)\rangle\rangle \in D\} \\
\mathrm{d}_{!x()}D &= \{g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleleft, n, *)\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleright, n, \_, \langle\dagger\rangle)\rangle\rangle \in D\} \\
\mathrm{d}_{?x[x_1,x_2]}D &= \{g \cdot \langle\gamma \cdot \langle x_1 \mapsto (\triangleright, n, a_1, a_2)\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleleft, n, \langle a_1, a_2\rangle)\rangle\rangle \in D\} \\
\mathrm{d}_{!x(x_1,x_2)}D &= \left\{ g \cdot \langle\gamma \cdot \langle x_1 \mapsto (\triangleleft, n, N_1, a_1, N_2, a_2)\rangle\rangle \;\middle|\; \begin{array}{l} g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleright, n, N, \langle a_1, a_2\rangle)\rangle\rangle \in D, \\ N_i = \{\mathrm{id}(b_i) \mid (\triangleleft, m, \langle b_1, b_2\rangle) \in \gamma, m \in N\} \end{array} \right\} \\
\mathrm{d}_{x \leftrightarrow y}D &= \{g \mid g \cdot \langle\langle x \mapsto a, y \mapsto a'\rangle\rangle \in D, a \Leftrightarrow a'\} \\
\mathrm{d}_{(l \| l')}D &= \left\{ g' \cdot h' \;\middle|\; g \cdot h \in D, g' \in \mathrm{d}_l\{g\}, \text{and } h' \in \mathrm{d}_{l'}\{h\} \right\} \\
\mathrm{d}_{\tau}D &= D
\end{aligned}
$$

Likewise, we define antiderivation reversing the reasoning above.

Definition 5.3 (Antiderivatives). For D a set of denotations, the antiderivative $\int_l D$ of D w.r.t. an action l is the set defined by:

$$
\begin{aligned}
\textstyle\int_{x[]}D &= \{g \cdot \langle\langle x \mapsto (\triangleleft, n, *)\rangle\rangle \mid g \in D\} \\
\textstyle\int_{x()}D &= \{g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleright, n, *)\rangle\rangle \mid g \cdot \langle\gamma\rangle \in D\} \\
\textstyle\int_{x[y]}D &= \{g \cdot \langle\gamma \cdot \delta \cdot \langle x \mapsto (\triangleleft, n, \mathrm{ids}(\gamma), a, \mathrm{ids}(\delta), b)\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto b\rangle, \delta \cdot \langle y \mapsto a\rangle\rangle \in D\} \\
\textstyle\int_{x(y)}D &= \{g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleright, n, a, b)\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto b, y \mapsto a\rangle\rangle \in D\} \\
\textstyle\int_{x \triangleleft \mathsf{inl}}D &= \{g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleleft, n, \mathsf{inl}, a)\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto a\rangle\rangle \in D\} \\
\textstyle\int_{x \triangleright \mathsf{inl}}D &= \{g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleright, n, \mathrm{id}(\gamma), \mathsf{inl}, a)\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto a\rangle\rangle \in D\} \\
\textstyle\int_{x \triangleleft \mathsf{inr}}D &= \{g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleleft, n, \mathsf{inr}, b)\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto b\rangle\rangle \in D\} \\
\textstyle\int_{x \triangleright \mathsf{inr}}D &= \{g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleright, n, \mathrm{id}(\gamma), \mathsf{inr}, b)\rangle\rangle \mid \langle\gamma \cdot \langle x \mapsto b\rangle\rangle \in D\} \\
\textstyle\int_{?x[y]}D &= \{g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleleft, n, \langle a\rangle)\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto a\rangle\rangle \in D\} \\
\textstyle\int_{!x(y)}D &= \{g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleright, n, \mathrm{id}(\gamma), \langle a\rangle)\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto a\rangle\rangle \in D\} \\
\textstyle\int_{?x[]}D &= \{g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleleft, n, \langle\dagger\rangle)\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleright, n, *)\rangle\rangle \in D\} \\
\textstyle\int_{!x()}D &= \{g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleright, n, \mathrm{id}(\gamma), \langle\dagger\rangle)\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleleft, n, *)\rangle\rangle \in D\} \\
\textstyle\int_{?x[x_1,x_2]}D &= \{g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleleft, n, \langle a_1, a_2\rangle)\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x_1 \mapsto (\triangleright, n, a_1, a_2)\rangle\rangle \in D\} \\
\textstyle\int_{!x(x_1,x_2)}D &= \{g \cdot \langle\gamma \cdot \langle x \mapsto (\triangleright, n, \mathrm{id}(\gamma), \langle a_1, a_2\rangle)\rangle\rangle \mid g \cdot \langle\gamma \cdot \langle x_1 \mapsto (\triangleleft, n, N_1, a_1, N_2, a_2)\rangle\rangle \in D\}
\end{aligned}
$$

$$\int_{x\leftrightarrow y} D = \left\{ g \cdot \langle\langle x \mapsto a, y \mapsto a^{\perp}\rangle\rangle \,\middle|\, g \in D \right\}$$
$$\int_{(l \parallel l')} D = \left\{ g \cdot h \,\middle|\, g' \cdot h' \in D, g \in \int_{l}\{g'\}, \text{and } h \in \int_{l'}\{h'\} \right\}$$
$$\int_{\tau} D = D$$

Interpretation of typed processes. We use sets of denotations for hyperenvironments (D) as denotations for well-typed processes. We define the denotational semantics of a well-typed process P as the subset $[\![P]\!]$ of $\bigcup_{P \vdash \mathcal{G}} \{\!|\mathcal{G}|\!\}$ given below by recursion on the structure of P.

Denotations for links are all pairs of dual name denotations.

$$[\![x{\leftrightarrow}y]\!] = \{\langle\langle x \mapsto a, y \mapsto a'\rangle\rangle \mid a \Leftrightarrow a'\}$$

Denotations for restriction are obtained by removing the restricted names and requiring that their observations are dual of each other.

$$[\![(\nu xy)P]\!] = \{g \cdot \langle\gamma \cdot \delta\rangle \mid g \cdot \langle\gamma \cdot \langle x \mapsto a\rangle, \delta \cdot \langle y \mapsto a'\rangle\rangle \in [\![P]\!] \text{ and } a \Leftrightarrow a'\}$$

Denotations for parallel composition reflect the denotation domain construction; they are given by cartesian product (restricted to pairs with disjoint identifiers since we assume well-formedness) and its unit, respectively.

$$[\![P \mid Q]\!] = [\![P]\!] \times [\![Q]\!] \qquad [\![\mathbf{0} \vdash \varnothing]\!] = \{\langle\rangle\}$$

Denotations for choice are the antiderivatives of denotations for each branch.

$$[\![x \triangleright \{\mathsf{inl} : P; \mathsf{inr} : Q\}]\!] = \int_{x \triangleright \mathsf{inl}} [\![P]\!] + \int_{x \triangleright \mathsf{inr}} [\![Q]\!]$$

All remaining terms are action prefixes. For those, denotations are given by the associated antiderivatives of the denotation of the continuation—with some care for the cases of $?x[]$, $?x[x_1, x_2]$, $!x(x')$, which have additional operations for managing server dependencies. For conciseness, let π range over prefixes except $?x[]$, $?x[x_1, x_2]$, $!x(x')$ in the following definitions.

$$[\![\pi.P]\!] = \int_{\pi} [\![P]\!] \qquad [\![?x[].P]\!] = \int_{?x[]} [\![x().P]\!] \qquad [\![?x[x_1, x_2].P]\!] = \int_{?x[x_1,x_2]} [\![x_1(x_2).P]\!]$$

Servers are the only prefix with more than one axiom yielding three cases.

$$\begin{aligned} [\![!x(x').P]\!] \;&=\; \int_{!x(x')} [\![P]\!] + \int_{!x()} [\![?z_1[].\cdots?z_n[].x[].\mathbf{0}]\!] + \\ &+\; \int_{!x(x_1,x_2)} [\![?z_1[z_1\sigma_1, z_1\sigma_2].\ldots.?z_n[z_n\sigma_1, z_n\sigma_2].x_1[x_2].(!x_1(x'\sigma_1).P_1 \mid !x_2(x'\sigma_2).P_2)]\!] \end{aligned}$$

for $\{x', z_1, \ldots, z_n\} = \mathrm{fn}(P)$ and any P_1 and P_2 such that $P_1 = P\sigma_1$, $P_2 = P\sigma_2$, and $\mathrm{fn}(P_1) \cap \mathrm{fn}(P_2) = \emptyset$.

Antiderivation also serves as a sanity check for the rule giving the denotations of links:

$$[\![x{\leftrightarrow}y]\!] = \int_{x\leftrightarrow y} [\![\mathbf{0}]\!].$$

Definition 5.4 (Denotational equivalence). Denotational equivalence $\eqsim$ is the relation on well-typed processes s.t. $P \eqsim Q$ whenever $[\![P]\!] = [\![Q]\!]$.

Denotational equivalence implies type equivalence.

Proposition 5.5. *If $P \eqsim Q$, then $P \vdash \mathcal{G}$ iff $Q \vdash \mathcal{G}$.*

Discussion. In general, our antiderivatives respect Fubini's theorem for double antiderivation [Fubini 1907]. This formalises the intuition that the order of independent actions is not discriminated.

Theorem 5.6 (Order-invariance). *For l_x, l_y, and P s.t. $x *_P y$ $\int_{l_x}\int_{l_y} [\![P]\!] = \int_{l_y}\int_{l_x} [\![P]\!] = \int_{(l_x \parallel l_y)} [\![P]\!]$.*

Example 5.7. Consider the processes described in Example 2.1. The type of bits is $1 \oplus 1$. The terms introduced for bit transmission have the following denotations, parametric in those of P and Q (we omit types, since they do not matter for our argument here).

$$[\![x[0].P]\!] = \int_{x[y]} \int_{y \triangleleft \mathsf{inl}} \int_{y[]} [\![P]\!] = \{g \cdot \langle \gamma \cdot \langle x \mapsto (\triangleleft, \_, (\triangleleft, \_, \mathsf{inl}, (\triangleleft, \_, *)), a) \rangle \rangle \mid g \cdot \langle \gamma \cdot \langle x \mapsto a \rangle \rangle \in [\![P]\!]\}$$

$$[\![x[1].P]\!] = \int_{x[y]} \int_{y \triangleleft \mathsf{inr}} \int_{y[]} [\![P]\!] = \{g \cdot \langle \gamma \cdot \langle x \mapsto (\triangleleft, \_, (\triangleleft, \_, \mathsf{inr}, (\triangleleft, \_, *), a) \rangle \rangle \mid g \cdot \langle \gamma \cdot \langle x \mapsto a \rangle \rangle \in [\![P]\!]\}$$

$$[\![x \triangleright \{0 \mapsto P; 1 \mapsto Q\}]\!] = \int_{x \triangleleft \mathsf{inl}} [\![P]\!] + \int_{x \triangleleft \mathsf{inr}} [\![Q]\!]$$

Below is the semantics of a client sending 0 and 1:

$$[\![Client^{01}_{xz}]\!] = \left\{ \left\langle\!\left\langle \begin{array}{l} z \mapsto (\triangleleft, n, b, (\triangleleft, *)), x \mapsto (\triangleleft, \_, \langle (\triangleleft, \_, (\triangleleft, \_, \mathsf{inl}, (\triangleleft, *)), \\ (\triangleleft, \_, (\triangleleft, \_, \mathsf{inr}, (\triangleleft, *)), (\triangleright, \_, \{n\}, b, (\triangleright, *)))))\rangle \end{array} \right\rangle\!\right\rangle \;\middle|\; b \in \{\mathsf{inl}, \mathsf{inr}\} \right\}$$

Observe how the order of transmission of each bit is preserved in the structure of the denotation for x, that the reply b appears in the denotations of both x and z, and that the semantics includes all possible replies the client may receive. As a consequence, the semantics captures the fact that the client echoes the server reply, whatever that may be.

Remark 5.8. Consider $P_1 = ?x[x_1, x_2].?x_1[].?x_2[x'].Q$ and $P_2 = ?x[x_1, x_2].?x_1[x'].?x_2[].Q$. Our behavioural equivalences (bisimilarity, barbed congruence, and denotational equivalence) discriminate these two processes, because we can observe how each copy (x_1 and x_2) is used. We could have a coarser behavioural theory where copies of the same server are not distinguished, by adopting the following transition rules that non-deterministically assign the names x_1 and x_2 to the copies.

$$\frac{\{x_1, x_2\} = \{y_1, y_2\}}{?x[x_1, x_2].P \xrightarrow{?x[y_1, y_2]} y_1(y_2).P} \qquad \frac{?x[x_1, x_2].P \xrightarrow{l} y_1(y_2).P' \quad \{x_1, x_2\} = \{y_1, y_2\} \quad x \ast_{?x[x_1, x_2].P} \mathrm{fn}(l)}{?x[x_1, x_2].P \xrightarrow{(?x[y_1, y_2] \| l)} y_1(y_2).P'}$$

Likewise, it suffices to adopt trees with unordered branching in non-linear observability predicates and interpretations of exponentials in the denotational semantics—just regard $\langle a_1, a_2 \rangle$ as an unordered pair.

Remark 5.9. Atkey [2017] proposed a denotational semantics for Classical Processes (CP) [Wadler 2014] that is coarser than ours when it comes to observing client requests. We give a representative example. Consider the processes $P_1 = ?x[x'].Q$ and $P_2 = ?x[x_1, x_2].?x_1[].?x_2[x'].Q$. The difference is that P_1 uses a single instance of the server at x, whereas P_2 first duplicates the server at x and then uses only one of the two copies (x_2), discarding the other copy. In [Atkey 2017], P_1 and P_2 are observationally and denotationally equivalent. This makes sense because weakening and contraction do not have corresponding terms in CP (so P_1 and P_2 would essentially be syntactically equivalent). Differently, our explicit terms and their semantics clearly show that P_2 makes the context (the server) "work more" in HCP, which motivates the distinction between P_1 and P_2. If we wished for the same coarse equivalence as in [Atkey 2017], we could exploit delayed actions for cancelling out unnecessary duplications whenever the continuation would dispose of one of the copies, as in the following rules. These would align bisimilarity and barbed congruence with Atkey's interpretation of exponentials (at the cost of a more complex lts).

$$\frac{P \xrightarrow{?y[]} P' \quad y \in \{x_1, x_2\}}{?x[x_1, x_2].P \xrightarrow{\tau} P'}\,\mathrm{c1} \qquad \frac{P \xrightarrow{?y_i[]} \not\quad \{y_1, y_2\} = \{x_1, x_2\}}{?x[x_1, x_2].P \xrightarrow{?x[y_1, y_2]} y_1(y_2).P}\,\mathrm{c2}$$

6 FULL ABSTRACTION

Barbed congruence implies denotational equivalence: given the element g of $[\![P]\!]$, there is a family of contexts $\mathbb{C}$ with the property that $g \in [\![Q]\!]$ whenever $C[P]$ and $C[Q]$ satisfy the same observability predicates for every $C \in \mathbb{C}$.

Theorem 6.1. $\cong \subseteq \backsimeq$.

Recall from Section 5 that taking derivatives of denotations with respect to an observable yields the denotations that can follow that actions. This interpretation suggests to define a labelled transition system on process denotations where transitions correspond to derivation.

Definition 6.2. The lts of denotations has as state-space the image of $[\![-]\!]$ and as transition relation the relation $D \xrightarrow{l} \mathrm{d}_l D$.

There is an operational correspondence between the lts of processes and that of denotations.

Lemma 6.3. *Let P be well-typed. Then:*

- *If $P \xrightarrow{l} P'$, then $[\![P]\!] \xrightarrow{l} [\![P']\!]$.*
- *If $[\![P]\!] \xrightarrow{l} [\![Q]\!]$, then $P \xRightarrow{l} P'$ for some $P' \backsimeq Q$.*

As a consequence, a process and its denotation are bisimilar and hence denotationally equivalent processes are bisimilar.

Theorem 6.4. $\backsimeq \subseteq \approx$.

By Theorems 4.14, 6.1 and 6.4, all three observational equivalences agree on well-typed programs.

Theorem 6.5 (Full abstraction). *For well-typed processes, $\approx = \cong = \backsimeq$.*

7 RELATED WORK

HCP stands on the shoulders of the prior work by Wadler [2014] on Classical Processes (CP), which built on the works by Caires and Pfenning [2010], Abramsky [1994], and Bellin and Scott [1994] on "Proofs as Processes". We have already discussed the distinctive points of HCP in the Introduction.

Pérez et al. [2014] introduced *typed context bisimilarity* as an observational equivalence for processes typed with intuitionistic linear logic [Caires and Pfenning 2010]. The idea is to name one of the free channels of a process as "the" representative channel and the others as "requirements". Then, typed context bisimilarity equates two processes if they perform the same actions on the representative channel once all their requirements are provided (using contexts). By contrast, our bisimilarity is standard: it does not require types or any distinction of the roles of channels, and it does not require reasoning on contexts. Also, we validated it with full abstraction.

Atkey [2017] explored the first notion of observation for CP. To cope with the lack of an lts, his contexts for contextual equivalence are not processes, but rather special configuration terms designed to make communication reductions visible. Our approach is simpler, because we can just look at transitions with observable actions using our lts. Studying the behavioural theory of HCP does not require configurations, and we showed how a fully-abstract denotational semantics can be formulated based on delayed actions.

In the original presentation of delayed actions by Merro and Sangiorgi [2004], normal (non-delayed) action prefixes are syntactically distinguished from delayed action prefixes (using the prefixing symbol ":" instead of "."). The same distinction could be introduced to HCP, adding extra machinery on top of our proof theory to preserve full abstraction. Namely, we would need to support enforcing sequentiality between independent channels at the same process. This could be

done by extending HCP with safe circular dependencies, *e.g.*, following the studies by [Carbone et al. 2017] and [Dardha and Gay 2018]. We chose our formulation for economy of the calculus.

In connection to our delayed actions, Bellin and Scott [1994] formulated syntactic permutations of independent actions to match permutations of rule applications in linear logic. Later, DeYoung et al. [2012] proposed to leverage commuting conversions in intuitionistic linear logic by assigning asynchronous process terms to proofs, but this makes terms even more compound—their term for output (corresponding to rule ⊗), in our syntax, is $x[y] \mid P \mid Q$, denoting that continuations may perform some actions before $x[y]$ is executed.

8 CONCLUSIONS AND FUTURE WORK

The keystone of process algebras is the parallel operator $P \mid Q$, but up to this work the connection between parallel and linear logic was indirect—no proof-theoretical inference rule was a straight match for parallel composition, but rather included parallel as part of more complex terms. The consequence for the agenda of "Proofs as Processes" [Abramsky 1994]: either accept that processes have sound structures and equivalences that cannot be captured by the proof theory, as done by Caires and Pfenning [2010], or give up on parallel as an independent operator, as done by Wadler [2014]. HCP cuts the head off the snake by using hyperenvironments to represent parallelism in sequents, internalised by the connective ⊗. We can now have our cake and eat it too: the semantics of well-typed processes is completely captured by a proof theory rooted in linear logic, and $P \mid Q$ is an operator in its own right respecting the expected equational laws. As a first step in taking advantage of this unification, we linked the ideas developed by Atkey [2017] for a denotational semantics for Classical Processes (CP) [Wadler 2014] to the standard theory of bisimilarity.

The design of HCP focused on the basic features of CP. In the future, we intend to study principled extensions that allow for capturing more behaviours.

Lindley and Morris [2016] extended CP with recursive types and their accompanying reductions. Adding the corresponding transitions to our lts seems straightforward, and our approach with derivatives to the definition of denotations might offer insight in how the denotational semantics of HCP can be extended to capture recursion. Similar considerations apply for non-determinism, introduced to intuitionistic linear logic by Caires and Pérez [2017].

Another interesting feature would be process mobility, *i.e.* the ability to communicate code. Toninho et al. [2013] obtained this for intuitionistic linear logic using a monadic integration with a functional programming model. Montesi [2018] extended CP with higher-order communications directly, by using environments as types for process variables. Thus, the obvious starting point to extend HCP to process mobility would be to type process variables with hyperenvironments. Historically, constructing tractable and characteristic behavioural equivalences for higher-order process calculi has been nontrivial in general [Sangiorgi et al. 2011]. The lts of HCP (extended to higher-order communications) might be helpful in this regard, because we can adapt bisimilarity notions from studies on session types [Kouzapas et al. 2017].

One of the most important extensions of session types is Multiparty Session Types, by Honda et al. [2016]. Carbone et al. [2017] showed that multiparty session types can be captured in CP by generalising the notion of duality to *coherence*, which checks for the compatibility of multiple types. This yields a generalised cut rule that can compose many processes instead of two. In HCP, this would correspond to extending rule H-CUT to the cutting of multiple environments.

ACKNOWLEDGMENTS

We are grateful to P. Wadler, S. Lindley, and the anonymous reviewers for their comments.This work was partially supported by the Independent Research Fund Denmark, grant no. DFF-7014-00041, and the Engineering and Physical Sciences Research Council, grant no. EP/L01503X/1.

REFERENCES

Samson Abramsky. 1994. Proofs as Processes. *Theor. Comput. Sci.* 135, 1 (1994), 5–9.

Robert Atkey. 2017. Observed Communication Semantics for Classical Processes. In *ESOP (Lecture Notes in Computer Science)*, Vol. 10201. Springer, 56–82.

Robert Atkey, Sam Lindley, and J. Garrett Morris. 2016. Conflation Confers Concurrency. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.), Vol. 9600. Springer, 32–55. https://doi.org/10.1007/978-3-319-30936-1_2

Arnon Avron. 1991. Hypersequents, logical consequence and intermediate logics for concurrency. *Ann. Math. Artif. Intell.* 4 (1991), 225–248.

Michael Barr. 1991. *-Autonomous Categories and Linear Logic. *Mathematical Structures in Computer Science* 1 (1991), 159–178.

Michael Barr. 1996. *-Autonomous categories, revisited. *Journal of Pure and Applied Algebra* 111, 1 (1996), 1 – 20. https://doi.org/10.1016/0022-4049(95)00040-2

Gianluigi Bellin and Philip J. Scott. 1994. On the pi-Calculus and Linear Logic. *TCS* 135, 1 (1994), 11–65.

Filippo Bonchi, Marcello M. Bonsangue, Helle Hvid Hansen, Prakash Panangaden, Jan J. M. M. Rutten, and Alexandra Silva. 2014. Algebra-coalgebra duality in Brzozowski's minimization algorithm. *ACM Trans. Comput. Log.* 15, 1 (2014), 3:1–3:29.

Tomasz Brengos, Marino Miculan, and Marco Peressotti. 2015. Behavioural equivalences for coalgebras with unobservable moves. *JLAMP* 84, 6 (2015), 826–852.

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *JACM* 11, 4 (Oct. 1964), 481–494. https://doi.org/10.1145/321239.321249

Luís Caires and Jorge A. Pérez. 2017. Linearity, Control Effects, and Behavioral Types. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science)*, Hongseok Yang (Ed.), Vol. 10201. Springer, 229–259. https://doi.org/10.1007/978-3-662-54434-1_9

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR*. 222–236.

Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. 2016. Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types. In *CONCUR (LIPIcs)*, Vol. 59. 33:1–33:15.

Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. 2017. Multiparty session types as coherence proofs. *Acta Informatica* (2017). https://doi.org/10.1007/s00236-016-0285-y Also: CONCUR, pages 412–426, 2015.

Ornela Dardha and Simon J. Gay. 2018. A New Linear Logic for Deadlock-Free Session-Typed Processes. In *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science)*, Christel Baier and Ugo Dal Lago (Eds.), Vol. 10803. Springer, 91–109. https://doi.org/10.1007/978-3-319-89366-2_5

Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France (LIPIcs)*, Patrick Cégielski and Arnaud Durand (Eds.), Vol. 16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 228–242. https://doi.org/10.4230/LIPIcs.CSL.2012.228

G. Fubini. 1907. Sugli integrali multipli. *Rom. Acc. L. Rend. (5)* 16, 1 (1907), 608–614.

Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50 (1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. 1998. Language primitives and type disciplines for structured communication-based programming. In *ESOP*. 22–138.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *JACM* 63, 1 (2016), 9. Also: POPL, 2008, pages 273–284.

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1999. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.* 21, 5 (1999), 914–947. https://doi.org/10.1145/330249.330251

Dimitrios Kouzapas, Jorge A. Pérez, and Nobuko Yoshida. 2017. Characteristic bisimulation for higher-order session processes. *Acta Inf.* 54, 3 (2017), 271–341. https://doi.org/10.1007/s00236-016-0289-7

Sam Lindley and Garrett Morris. 2016. Talking Bananas: structural recursion for session types. In *ICFP*. ACM. To appear.

Massimo Merro and Davide Sangiorgi. 2004. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science* 14, 5 (2004), 715–767.

Robin Milner. 1989. *Communication and Concurrency*. Prentice-Hall.

Fabrizio Montesi. 2018. Classical Higher-Order Processes. *CoRR* abs/1802.02917 (2018). arXiv:1802.02917 http://arxiv.org/abs/1802.02917

James H. Morris. 1968. *Lambda-calculus models of programming languages*. Ph.D. Dissertation. http://opac.inria.fr/record=b1000512 PHD.

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

Gordon D. Plotkin. 2004. A structural approach to operational semantics. *J. Log. Algebr. Program.* 60-61 (2004), 17–139.

Davide Sangiorgi. 1993. From pi-Calculus to Higher-Order pi-Calculus - and Back. In *TAPSOFT*. Springer, 151–166.

Davide Sangiorgi. 1996. Pi-Calculus, Internal Mobility, and Agent-Passing Calculi. *TCS* 167, 1&2 (1996), 235–274.

Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. 2011. Environmental bisimulations for higher-order languages. *ACM Trans. Program. Lang. Syst.* 33, 1 (2011), 5:1–5:69. https://doi.org/10.1145/1889997.1890002

Davide Sangiorgi and David Walker. 2001. *The Pi-Calculus - a theory of mobile processes.* Cambridge University Press.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *ESOP (Lecture Notes in Computer Science)*, Vol. 7792. Springer, 350–369.

Vasco T. Vasconcelos. 2012. Fundamentals of session types. *Inf. Comput.* 217 (2012), 52–70.

Philip Wadler. 2014. Propositions as sessions. *JFP* 24, 2–3 (2014), 384–418. Also: ICFP, pages 273–286, 2012.

Nobuko Yoshida, Kohei Honda, and Martin Berger. 2007. Linearity and bisimulation. *J. Log. Algebr. Program.* 72, 2 (2007), 207–238.