TOWARDS RACES IN LINEAR LOGIC

WEN KOKKE [a], J. GARRETT MORRIS [b], AND PHILIP WADLER [a]

[a] University of Edinburgh, Edinburgh, UK
    e-mail address: {wen.kokke,wadler}@ed.ac.uk

[b] University of Kansas, Lawrence, KS, USA
    e-mail address: garrett@ittc.ku.edu

ABSTRACT. Process calculi based in logic, such as πDILL and CP, provide a foundation for deadlock-free concurrent programming, but exclude non-determinism and races. HCP is a reformulation of CP which addresses a fundamental shortcoming: the fundamental operator for parallel composition from the π-calculus does not correspond to any rule of linear logic, and therefore not to any term construct in CP.

We introduce HCP_{ND}, which extends HCP with a novel account of non-determinism. Our approach draws on bounded linear logic to provide a strongly-typed account of standard process calculus expressions of non-determinism. We show that our extension is expressive enough to capture many uses of non-determinism in untyped calculi, such as non-deterministic choice, while preserving HCP's meta-theoretic properties, including deadlock freedom.

1. INTRODUCTION

Consider the following scenario:

> Ami and Boé are working from home one morning when they each get a craving for a slice of cake. Being denizens of the web, they quickly find the nearest store which does home deliveries. Unfortunately for them, they both order their cake at the *same* store, which has only one slice left. After that, all it can deliver is disappointment.

This is an example of a *race condition*. We can model this scenario in the π-calculus, where 👦, 👩 and 🗓️ are processes modelling Ami, Boé and the store, and 🍰 and 🙌 are channels giving access to a slice of cake and disappointment, respectively. This process has two possible outcomes: either Ami gets the cake, and Boé gets disappointment, or vice versa.

$$(x(y).👦 \parallel x(z).👩 \parallel x\langle 🍰 \rangle.x\langle 🙌 \rangle.🗓️)$$

$$\Downarrow_*$$

$$(👦\{🍰/y\} \parallel 👩\{🙌/z\} \parallel 🗓️) \quad \text{or} \quad (🗓️ \parallel 👦\{🙌/y\} \parallel 👩\{🍰/z\})$$

While Ami or Boé may not like all of the outcomes, it is the store which is responsible for implementing the online delivery service, and the store is happy with either outcome. Thus, the above is an interaction we would like to be able to model.

Now consider another scenario, which takes place *after* Ami has already bought the cake:

> Boé is *really* disappointed when she finds out the cake has sold out. Ami, always looking to make some money, offers to sell the slice to her for a profit. Boé agrees to engage in a little bit of back-alley cake resale, but sadly there is no trust between the two. Ami demands payment first. Boé would rather get her slice of cake before she gives Ami the money.

This is an example of a *deadlock*. We can also model this scenario in the π-calculus, where 💷 is a channel giving access to some adequate amount of money.

$$(x(z).y\langle🍰\rangle.\,🧑 \parallel y(w).x\langle💷\rangle.\,👩) \quad \not\Rightarrow^\star$$

The above process does not reduce. As both Ami and Boé would prefer the exchange to be made, this interaction is desired by *neither*. Thus, the above is an interaction we would like to exclude.

Session types [Hon93] statically guarantee that concurrent programs, such as those above, respect communication protocols. Session-typed calculi with logical foundations, such as πDILL [CP10] and CP [Wad12], obtain deadlock freedom as a result of a close correspondence with logic. These systems, however, also rule out non-determinism and race conditions. In this paper, we demonstrate that logic-inspired type systems need not rule out races.

We present HCP_{ND}, an extension of HCP with a novel account of non-determinism and races. Inspired by bounded linear logic [GSS92], we introduce a form of shared channels in which the type of a shared channel tracks how many times it is reused. As in the untyped π-calculus, sharing introduces the potential for non-determinism. We show that our approach is sufficient to capture practical examples of races, such as an online store, as well as other formal characterizations of non-determinism, such as non-deterministic choice. However, HCP_{ND} does not lose the meta-theoretical benefits of HCP: we show that it enjoys termination and deadlock-freedom.

An important limitation of our work is that types in HCP_{ND} explicitly count the potential races on a channel. It works fine when there are two or three races, but not n for an arbitary n. The latter case is obviously important, and we see the main value of our work as a stepping stone to this more general case.

HCP_{ND} is based on HCP [KMP18, KMP19]. HCP is a reformulation of CP which addresses a fundamental shortcoming: the fundamental operator for parallel composition from the π-calculus does not correspond to any rule of linear logic, and therefore not to any term construct in CP.

This paper proceeds as follows. In section 2, we discuss recent approaches to non-determinism in logic-inspired session-typed process calculi. In section 3, we introduce a variant of CP and prove progress and preservation. In section 4, we introduce HCP_{ND}. In section 5, we discuss cuts with leftovers. In section 6, we discuss the relation to manifest sharing [BP17]. Finally, in section 7, we conclude with a discussion of the work done in this paper and potential avenues for future work.

Remark (Variants of HCP). *There are two variants of HCP: a version with delayed actions, introduced by Kokke, Montesi, and Peressotti [KMP19], and a version without delayed actions,*

introduced by Kokke, Montesi, and Peressotti [KMP18]. Delayed actions are not an essential part of HCP, but significantly complicate the theory. Therefore, we base our work on the variant without delayed actions. For typographical simplicity, we will refer to the system without delayed actions as HCP, instead of HCP⁻ . Should we need to refer to the system with delayed actions, we will use DHCP.

2. Non-determinism, Logic, and Session Types

Recent work extended πDILL and CP with operators for non-deterministic behaviour [ALM16, Cai14, CP17]. These extensions all implement an operator known as non-deterministic local choice. (This operator is written as $P + Q$, but should not be confused with input-guarded choice from the π-calculus [MPW92].) Non-deterministic local choice can be summarised by the following typing and reduction rules:

$$\frac{P \vdash \Gamma \qquad Q \vdash \Gamma}{P + Q \vdash \Gamma} \qquad\qquad \begin{array}{l} P + Q \Longrightarrow P \\ P + Q \Longrightarrow Q \end{array}$$

Local choice introduces non-determinism explicitly, by listing all possible choices. This is unlike the π-calculus, where non-determinism arises due to multiple processes communicating on shared channels. We can easily implement local choice in the π-calculus, using a nullary communication:

$$(x\langle\rangle.0 \parallel x().P \parallel x().Q)$$

$$\Downarrow_\star$$

$$(P \parallel x().Q) \quad \text{or} \quad (x().P \parallel Q)$$

In this implementation, the process $x\langle\rangle.0$ will "unlock" either P or Q, leaving the other process deadlocked. Or we could use input-guarded choice:

$$(x\langle\rangle.0 \parallel (x().P + x().Q))$$

However, there are many non-deterministic processes in the π-calculus that are awkward to encode using non-deterministic local choice. Let us recall our example:

$$(x\langle🍰\rangle.x\langle🙌\rangle.📅 \parallel x(y).🧑 \parallel x(z).👩)$$

$$\Downarrow_\star$$

$$(📅 \parallel 🧑\{🍰/y\} \parallel 👩\{🙌/z\}) \quad \text{or} \quad (📅 \parallel 🧑\{🙌/y\} \parallel 👩\{🍰/z\})$$

This non-deterministic interaction involves communication. If we wanted to write down a process which exhibited the same behaviour using non-deterministic local choice, we would have to write the following process:

$$(x\langle🍰\rangle.y\langle🙌\rangle.📅 \parallel x(z).🧑 \parallel y(w).👩) + (y\langle🍰\rangle.x\langle🙌\rangle.📅 \parallel x(z).🧑 \parallel y(w).👩)$$

$$\Downarrow_\star$$

$$(📅 \parallel 🧑\{🍰/y\} \parallel 👩\{🙌/z\}) \quad \text{or} \quad (📅 \parallel 🧑\{🙌/y\} \parallel 👩\{🍰/z\})$$

In essence, instead of modelling a non-deterministic interaction, we are enumerating the resulting deterministic interactions. This means non-deterministic local choice cannot model non-determinism in the way the π-calculus does. Enumerating all possible outcomes becomes worse the more processes are involved in an interaction. Imagine the following scenario:

> Three customers, Ami, Boé, and Cat, have a craving for cake. Should cake be sold out, however, well... a doughnut will do. They prepare to order their goods via an online store. Unfortunately, they all decide to use the same *shockingly* under-stocked store, which has only one slice of cake, and a single doughnut. After that, all it can deliver is disappointment.

We can model this scenario in the π-calculus, where 🧑, 🧑, 👩, and 🏪 are four processes modelling Ami, Boé, Cat, and the store, and 🍰, 🍩, and 🙌 are three channels giving access to a slice of cake, a so-so doughnut, and disappointment, respectively.

$$(x\langle🍰\rangle.x\langle🍩\rangle.x\langle🙌\rangle.🏪 \parallel x(y).🧑 \parallel x(z).👩 \parallel x(w).🧑)$$

$$\Downarrow_*$$

$$(🏪 \parallel 🧑\{🍰/y\} \parallel 👩\{🍩/z\} \parallel 🧑\{🙌/w\}) \text{ or } (🏪 \parallel 🧑\{🍰/y\} \parallel 👩\{🙌/z\} \parallel 🧑\{🍩/w\})$$

$$(🏪 \parallel 🧑\{🍩/y\} \parallel 👩\{🙌/z\} \parallel 🧑\{🍰/w\}) \text{ or } (🏪 \parallel 🧑\{🍩/y\} \parallel 👩\{🍰/z\} \parallel 🧑\{🙌/w\})$$

$$(🏪 \parallel 🧑\{🙌/y\} \parallel 👩\{🍰/z\} \parallel 🧑\{🍩/w\}) \text{ or } (🏪 \parallel 🧑\{🙌/y\} \parallel 👩\{🍩/z\} \parallel 🧑\{🍰/w\})$$

With the addition of one process, modelling Cat, we have increased the number of possible outcomes enormously! In general, the number of outcomes for these types of scenarios is $n!$, where n is the number of processes. This means that if we wish to translate any non-deterministic process to one using non-deterministic local choice, we can expect a factorial growth in the size of the term.

3. Hypersequent Classical Processes

In this section, we introduce HCP [KMP18], the basis for our calculus HCP_{ND}. The term language for HCP is a variant of the π-calculus [MPW92]. In HCP, processes (P, Q, R) communicate using names (x, y, z, ...). Each name is one of the two endpoints of a bidirectional communication channel [Vas12]. A channel is formed by connecting two endpoints using name restriction. This is in contrast to sections 1 and 2, where we used names to represent channels.

Definition 3.1 (Terms)**.**

$$
\begin{array}{llll}
P, Q, R & ::= & x \leftrightarrow y & \text{link} \\
& | & 0 & \text{terminated process} \\
& | & (\nu x x')P & \text{name restriction, "cut"} \\
& | & (P \parallel Q) & \text{parallel composition, "mix"} \\
& | & x[y].P & \text{output} \\
& | & x(y).P & \text{input} \\
& | & x[].P & \text{halt} \\
& | & x().P & \text{wait} \\
& | & x \triangleleft \mathtt{inl}.P & \text{select left choice} \\
& | & x \triangleleft \mathtt{inr}.P & \text{select right choice} \\
& | & x \triangleright \{\mathtt{inl} : P; \mathtt{inr} : Q\} & \text{offer binary choice} \\
& | & x \triangleright \{\} & \text{offer nullary choice}
\end{array}
$$

The variables x, y, z, u, v, and w range over channel endpoints. Occasionally, we use a, b, and c to range over *free* endpoints, *i.e.*, those which are not connected to another endpoint. The construct $x \leftrightarrow y$ links two endpoints [San96, Bor98], forwarding messages received on x

to y and vice versa. The construct $(\nu xx')P$ creates a new channel by connecting endpoints x and x'. By convention, we name dual endpoints using primes, *e.g.*, x and x'. However, the primes are merely a naming convention, and do not denote co-names, *e.g.*, x and x' are not inherently dual, only under a ν-binder $(\nu xx')$. The construct $P \parallel Q$ and composes two processes. In $x(y).P$ and $x[y].P$, round brackets denote input, square brackets denote output. We use bound output [San96], meaning that both input and output bind a new name.

Terms in HCP are identified up to structural congruence.

Definition 3.2 (Structural congruence)**.** The structural congruence $\equiv$ is the congruence closure over terms which satisfies the following additional axioms:

$$
\begin{array}{llll}
\text{SC-LinkSwap} & x{\leftrightarrow}y & \equiv & y{\leftrightarrow}x \\
\text{SC-ParComm} & P \parallel Q & \equiv & Q \parallel P \\
\text{SC-ParAssoc} & P \parallel (Q \parallel R) & \equiv & (P \parallel Q) \parallel R \\
\text{SC-ParNil} & P \parallel 0 & \equiv & P \\
\text{SC-ResNil} & (\nu xx')0 & \equiv & 0 \\
\text{SC-ResComm} & (\nu xx')(\nu yy')P & \equiv & (\nu yy')(\nu xx')P \\
\text{SC-ResExt} & (\nu xx')(P \parallel Q) & \equiv & P \parallel (\nu xx')Q \quad \text{if } x, x' \notin P
\end{array}
$$

Channels in HCP are typed using a session type system which is a conservative extension of linear logic.

Definition 3.3 (Types)**.**

$$
\begin{array}{rclll}
A, B, C & ::= & A \otimes B & \text{independent channels} & \mid \quad \mathbf{1} \quad \text{unit for } \otimes \\
& \mid & A \,\invamp\, B & \text{interdependent channels} & \mid \quad \bot \quad \text{unit for } \invamp \\
& \mid & A \oplus B & \text{internal choice} & \mid \quad \mathbf{0} \quad \text{unit for } \oplus \\
& \mid & A \,\&\, B & \text{external choice} & \mid \quad \top \quad \text{unit for } \&
\end{array}
$$

Duality plays a crucial role in both linear logic and session types. In HCP, the two endpoints of a channel are assigned dual types. This ensures that, for instance, whenever a process *sends* across a channel, the process on the other end of that channel is waiting to *receive*. Each type A has a dual, written $A^\perp$. Duality $(\cdot^\perp)$ is an involutive function on types.

Definition 3.4 (Duality)**.**

$$
\begin{array}{llll}
(A \otimes B)^\perp = A^\perp \invamp B^\perp & \mathbf{1}^\perp = \bot & (A \invamp B)^\perp = A^\perp \otimes B^\perp & \bot^\perp = \mathbf{1} \\
(A \oplus B)^\perp = A^\perp \,\&\, B^\perp & \mathbf{0}^\perp = \top & (A \,\&\, B)^\perp = A^\perp \oplus B^\perp & \top^\perp = \mathbf{0}
\end{array}
$$

Environments associate channels with types. Names in environments must be unique, and environments Γ and Δ can only be combined (Γ, Δ) if $\mathrm{cn}(\Gamma) \cap \mathrm{cn}(\Delta) = \varnothing$, where $\mathrm{cn}(\Gamma)$ denotes the set of channel names in Γ.

Definition 3.5 (Environments)**.** $\Gamma, \Delta, \Theta ::= x_1 : A_1 \ldots x_n : A_n$

HCP registers parallelism using hyper-environments. A hyper-environment is a multiset of environments. While names within environments must be unique, names may be shared between multiple environments in a hyper-environment. We write $\mathcal{G} \parallel \mathcal{H}$ to combine two hyper-environments.

Definition 3.6 (Hyper-environments)**.** $\mathcal{G}, \mathcal{H} ::= \varnothing \mid \mathcal{G} \parallel \Gamma$

Typing judgements associate processes with collections of typed channels.

Definition 3.7 (Typing judgements). A typing judgement $P \vdash \Gamma_1 \parallel \ldots \parallel \Gamma_n$ denotes that the process P consists of n independent, but potentially entangled processes, each of which communicates according to its own protocol Γ_i. Typing judgements can be constructed using the inference rules below.

Structural rules

$$\frac{}{x \leftrightarrow y \vdash x : A, y : A^{\perp}} \text{Ax} \qquad \frac{P \vdash \mathcal{G} \parallel \Gamma, x : A \parallel \Delta, x' : A^{\perp}}{(\nu x x')P \vdash \mathcal{G} \parallel \Gamma, \Delta} \text{Cut}$$

$$\frac{P \vdash \mathcal{G} \qquad Q \vdash \mathcal{H}}{P \parallel Q \vdash \mathcal{G} \parallel \mathcal{H}} \text{H-Mix} \qquad \frac{}{0 \vdash \varnothing} \text{H-Mix}_0$$

Logical rules

$$\frac{P \vdash \Gamma, y : A \parallel \Delta, x : B}{x[y].P \vdash \Gamma, \Delta, x : A \otimes B} \otimes \qquad \frac{P \vdash \Gamma, y : A, x : B}{x(y).P \vdash \Gamma, x : A \,\otimes\, B} (\otimes)$$

$$\frac{P \vdash \varnothing}{x[].P \vdash x : \mathbf{1}} \mathbf{1} \qquad \frac{P \vdash \Gamma}{x().P \vdash \Gamma, x : \perp} (\perp)$$

$$\frac{P \vdash \Gamma, x : A}{x \triangleleft \mathtt{inl}.P \vdash \Gamma, x : A \oplus B} (\oplus_1) \qquad \frac{P \vdash \Gamma, x : B}{x \triangleleft \mathtt{inr}.P \vdash \Gamma, x : A \oplus B} (\oplus_2)$$

$$\frac{P \vdash \Gamma, x : A \qquad Q \vdash \Gamma, x : B}{x \triangleright \{\mathtt{inl} : P; \mathtt{inr} : Q\} \vdash \Gamma, x : A \,\&\, B} (\&)$$

$$\text{(no rule for } \mathbf{0}) \quad \frac{}{x \triangleright \{\} \vdash \Gamma, x : \top} (\top)$$

Alternative syntax. In ($\mathbf{1}$), the only well-typed continuation P is the terminated process $\mathbf{0}$. We could use an alternative formulation of the rule, which combines ($\mathbf{1}$) and H-Mix$_0$. However, as H-Mix$_0$ is used on its own, and not just in combination with ($\mathbf{1}$), we chose the present formulation to avoid having multiple different representations of the terminated process in the language.

Reductions relate processes with their reduced forms.

Definition 3.8 (Reduction). Reductions are described by the smallest relation $\Longrightarrow$ on process terms closed under the rules below:

E-Link	$(\nu x x')(w \leftrightarrow x \parallel P)$	$\Longrightarrow$	$P\{w/x'\}$
E-Send	$(\nu x x')(x[y].P \parallel x'(y').R)$	$\Longrightarrow$	$(\nu x x')(\nu y y')(P \parallel R)$
E-Close	$(\nu x x')(x[].P \parallel x'().Q)$	$\Longrightarrow$	$P \parallel Q$
E-Sel$_1$	$(\nu x x')(x \triangleleft \mathtt{inl}.P \parallel x' \triangleright \{\mathtt{inl} : Q; \mathtt{inr} : R\})$	$\Longrightarrow$	$(\nu x x')(P \parallel Q)$
E-Sel$_2$	$(\nu x x')(x \triangleleft \mathtt{inr}.P \parallel x' \triangleright \{\mathtt{inl} : Q; \mathtt{inr} : R\})$	$\Longrightarrow$	$(\nu x x')(P \parallel R)$

$$\frac{P \Longrightarrow P'}{(\nu x x')P \Longrightarrow (\nu x x')P'} \text{E-LiftRes} \qquad \frac{P \Longrightarrow P'}{P \parallel Q \Longrightarrow P' \parallel Q} \text{E-LiftPar}$$

$$\frac{P \equiv Q \qquad Q \Longrightarrow Q' \qquad Q' \equiv P'}{P \Longrightarrow P'} \text{E-LiftSC}$$

We define unbound output in terms of bound output and link [LM15]:

$$x\langle y \rangle.P \triangleq x[z].(y \leftrightarrow z \parallel P)$$

$$\frac{P \vdash \Gamma, x : B}{x\langle y \rangle.P \vdash \Gamma, x : A \otimes B, y : A^{\perp}} \qquad \begin{array}{l} (\nu x x')(x\langle y \rangle.P \parallel x'(y').Q) \\ \Longrightarrow (\nu x x')(P \parallel Q\{y/y'\}) \end{array}$$

3.1. **Example.** HCP uses hyper-sequents to structure communication, and it is this structure which rules out deadlocked interactions. Let us go back to our example of a deadlocked interaction from section 1. If we want to type this interaction in HCP, we run into a problem: to connect x and y, and z and w, such that we get a deadlock, we need to construct the following term:

$$(\nu xx')(\nu yy')(x(u).y\langle 🍰 \rangle.👨 \parallel y'(v).x'\langle 💶 \rangle.👩).$$

However, there is no typing derivation for this term. We can construct a typing derivation down to the sequent below, but we cannot introduce *both* name restrictions: the CUT rule eliminates a hypersequent separator, which ensures that it only ever connects two independent processes, but the sequent below only has *one*.

$$x(z).y\langle 🍰 \rangle.👨 \parallel y'(w).x'\langle 💶 \rangle.👩 \vdash \begin{array}{l} x : \overline{💲}^{\perp} ⅋ \perp, y : 🎂 \otimes 1, 🍰 : 🎂^{\perp} \parallel \\ x' : \overline{💲} \otimes 1, y' : 🎂^{\perp} ⅋ \perp, 💶 : \overline{💲}^{\perp} \end{array}$$

3.2. **Metatheory.** HCP enjoys subject reduction, termination, and progress [KMP18].

Lemma 3.9 (Preservation for $\equiv$). *If $P \equiv Q$, then $P \vdash \mathcal{G}$ iff $Q \vdash \mathcal{G}$.*

Proof. By induction on the derivation of $P \equiv Q$. $\qquad\square$

Theorem 3.10 (Preservation). *If $P \vdash \mathcal{G}$ and $P \Longrightarrow Q$, then $Q \vdash \mathcal{G}$.*

Proof. By induction on the derivation of $P \Longrightarrow Q$. $\qquad\square$

Definition 3.11 (Actions). A process P acts on x whenever x is free in the outermost term constructor of P, *e.g.*, $x[y].P$ acts on x but not on y, and $x \leftrightarrow y$ acts on both x and y. A process P is an action if it acts on some channel x.

Definition 3.12 (Canonical forms). A process P is in canonical form if

$$P \equiv (\nu x_1 x_1') \ldots (\nu x_n x_n')(P_1 \mid \cdots \mid P_{n+m+1}),$$

such that: no process P_i is a cut or a mix; no process P_i is a link acting on a bound channel x_i or x_i'; and no two processes P_i and P_j are acting on dual endpoints x_i and x_i' of the same channel.

Lemma 3.13. *If a well-typed process P is in canonical form, then it is blocked on an external communication, i.e., $P \equiv (\nu x_1 x_1') \ldots (\nu x_n x_n')(P_1 \mid \cdots \mid P_{n+m+1})$ such that at least one process P_i acts on a free name.*

Proof. We have $P \equiv (\nu x_1 x_1') \ldots (\nu x_n x_n')(P_1 \parallel \ldots \parallel P_{n+m+1})$, such that no P_i is a cut or a link acting on a bound channel, and no two processes P_i and P_j are acting on the endpoints of the same channel. The prefix of cuts and mixes introduces n channels. Each application of cut requires an application of mix, so the prefix introduces $n+m+1$ processes. Therefore, at least $m+1$ of the processes P_i must be acting on a free channel, i.e., blocked on an external communication. $\qquad\square$

Theorem 3.14 (Progress). *If $P \vdash \mathcal{G}$, then either P is in canonical form, or there exists a process Q such that $P \Longrightarrow Q$.*

Proof. We consider the maximum prefix of cuts and mixes of P such that

$$P \equiv (\nu x_1 x_1') \ldots (\nu x_n x_n')(P_1 \parallel \ldots \parallel P_{n+m+1}),$$

and no P_i is a cut. If any process P_i is a link, we reduce by $(\leftrightarrow)$. If any two processes P_i and P_j are acting on dual endpoints x_i and x_i' of the same channel, we rewrite by $\equiv$ and reduce by the appropriate β-rule. Otherwise, P is in canonical form. $\qquad\square$

Theorem 3.15 (Termination). *If $P \vdash \mathcal{G}$, then there are no infinite $\Longrightarrow$-reduction sequences.*

Proof. Every reduction reduces a single cut to zero, one or two cuts. However, each of these cuts is smaller, measured in the size of the cut formula. Furthermore, each instance of the structural congruence preserves the size of the cut. Therefore, there cannot be an infinite $\Longrightarrow$-reduction sequence. $\qquad\square$

3.3. **Erratum for HCP.** The typing rules for HCP presented here are more restrictive than those in earlier publications [KMP18]. Progress does not hold for the earlier version. For instance, the following process is stuck, yet typeable:

$$\cfrac{\cfrac{\cfrac{\cfrac{\overline{0 \vdash \varnothing}}{y[].0 \vdash y:\mathbf{1}}}{x[].y[].0 \vdash x:\mathbf{1} \parallel y:\mathbf{1}} \quad \cfrac{\overline{x\leftrightarrow z \vdash x:\bot, z:\mathbf{1}}}{y().x\leftrightarrow z \vdash x:\bot, y:\bot, z:\mathbf{1}}}{\cfrac{x[].y[].0 \parallel y().x\leftrightarrow z \vdash x:\mathbf{1} \parallel y:\mathbf{1} \parallel x:\bot, y:\bot, z:\mathbf{1}}{\cfrac{(\nu y)(x[].y[].0 \parallel y().x\leftrightarrow z) \vdash x:\mathbf{1} \parallel x:\bot, z:\mathbf{1}}{(\nu x)(\nu y)(x[].y[].0 \parallel y().x\leftrightarrow z) \vdash z:\mathbf{1}}}}$$

The earlier typing rules failed to guarantee a crucial property: each typing environment should correspond to one top-level action. The rules presented in this paper fixes the problem by disallowing hyper-environments in logical rules.

The move from channel names to endpoint names is not essential to the fix, but significantly streamlines the presentation. Otherwise, the type system must guarantee that each channel name occurs at most twice in the hypersequent, and if twice, then with dual types. Using endpoint names, it is sufficient to require that all names be distinct.

4. SHARED CHANNELS AND NON-DETERMINISM

In this section, we will discuss our main contribution: an extension of HCP which allows for races while still excluding deadlocks. We have seen in section 3.1 how HCP excludes deadlocks, but how exactly does HCP exclude races? Let us return to our example in π-calculus from section 1, to the interaction between Ami, Boé and the store.

$$(x\langle \text{🍰}\rangle.x\langle\text{🙌}\rangle.\text{📇} \parallel x(y).\text{🧑} \parallel x(z).\text{👩})$$

$$\Downarrow_{*}$$

$$(\text{📇} \parallel \text{🧑}\{\text{🍰}/y\} \parallel \text{👩}\{\text{🙌}/z\}) \quad \text{or} \quad (\text{📇} \parallel \text{🧑}\{\text{🙌}/y\} \parallel \text{👩}\{\text{🍰}/z\})$$

Races occur when more than two processes attempt to communicate simultaneously over the *same* channel. However, the CUT rule of HCP requires that *exactly two* processes communicate over each channel:

$$\dfrac{P \;\vdash\; \mathcal{G} \parallel \Gamma, x\!:\!A \parallel \Delta, x'\!:\!A^{\perp}}{(\nu xx')P \;\vdash\; \mathcal{G} \parallel \Gamma, \Delta} \;\textsc{Cut}$$

We could attempt to write down a protocol for our example, stating that the store has a pair of channels $x, y :$ 🎂 with which it communicates with Ami and Boé, taking 🎂 to be the type of interactions in which cake *may* be obtained, i.e. of both 🍰 and 🙌, and state that the store communicates with Ami *and* Boé over a channel of type 🎂 $\otimes$ 🎂. However, this *only* models interactions such as the following:

$$\dfrac{\dfrac{\dfrac{\dfrac{🧑 \vdash \Gamma, y\!:\!🎂^{\perp} \qquad 🧑 \vdash \Delta, x\!:\!🎂^{\perp}}{(🧑 \parallel 🧑) \vdash \Gamma, y\!:\!🎂^{\perp} \parallel \Delta, x\!:\!🎂^{\perp}} \;\textsc{H-Mix}}{x[y].(🧑 \parallel 🧑) \vdash \Gamma, \Delta, x\!:\!🎂^{\perp} \otimes 🎂^{\perp}} \;(\otimes) \qquad \dfrac{\dfrac{🏪 \vdash \Theta, y'\!:\!🎂, x'\!:\!🎂}{x'(y').🏪 \vdash \Theta, x'\!:\!🎂 ⅋ 🎂} \;(⅋)}{}}{(x[y].(🧑 \parallel 🧑) \parallel x'(y').🏪) \vdash \Gamma, \Delta, x\!:\!🎂^{\perp} \otimes 🎂^{\perp} \parallel \Theta, x'\!:\!🎂 ⅋ 🎂} \;\textsc{H-Mix}}{(\nu xx')(x[y].(🧑 \parallel 🧑) \parallel x'(y').🏪) \vdash \Gamma, \Delta, \Theta} \;\textsc{Cut}$$

In this interaction, Ami will get whatever the store decides to send on x, and Boé will get whatever the store decides to send on y. This means that this interactions gives the choice of who receives what *to the store*. This is not an accurate model of our original example, where the choice of who receives the cake is non-deterministic and depends on factors outside of any of the participants' control!

Modelling racy behaviour, such as that in our example, is essential to describing the interactions that take place in realistic concurrent systems. We would like to extend HCP to allow such races in a way which mirrors the way in which the π-calculus handles non-determinism. Let us return to our example:

$$(x\langle 🍰\rangle.x\langle 🙌\rangle.🏪 \parallel x(y).🧑 \parallel x(z).🧑)$$

In this interaction, we see that the channel x is only used as a way to connect the various clients, Ami and Boé, to the store. The *real* communication, sending the slice of cake and disappointment, takes places on the channels 🍰, 🙌, y and z. Inspired by this, we add two new constructs to the term language of HCP for sending and receiving on a *shared* channel. These actions are marked with a $\star$ to distinguish them from ordinary sending and receiving.

Definition 4.1 (Terms)**.** We extend theorem 3.1 as follows:

$$P, Q, R ::= \ldots$$
$$\mid \star x[y].P \quad \text{client creation}$$
$$\mid \star x(y).P \quad \text{server interaction}$$

As before, round brackets denote input, square brackets denote output. Note that $\star x[y].P$, much like $x[y].P$, is a bound output: both client creation and server interaction bind a new name. The structural congruence, which identifies certain terms, is the same as theorem 3.2.

In any non-deadlock interaction between a server and some clients, there must be *exactly* as many clients as there are server interactions. Therefore, we add two new *dual* types for client pools and servers, which track how many clients or server interactions they represent.

Definition 4.2 (Types)**.** We extend theorem 3.3 as follows:

$$A, B, C \quad ::= \quad \ldots$$
$$\mid \quad !_n A \quad \text{pool of } n \text{ clients}$$
$$\mid \quad ?_n A \quad n \text{ server interactions}$$

The types $!_nA$ and $?_nA^\perp$ are dual. (The subscripts must be identical, *i.e.*, $!_nA$ is not dual to $?_mA^\perp$ if $n \neq m$.) Duality remains an involutive function.

We have to add typing rules to associate our new client and server interactions with their types. The definition for environments will remain unchanged, but we will extend the definition for the typing judgement. To determine the new typing rules, we essentially answer the question "What typing constructs do we need to complete the following proof?"

$$
\begin{array}{ccc}
\text{👦} \vdash \Gamma, x' : \text{🎂}^\perp & \text{👩} \vdash \Delta, y' : \text{🎂}^\perp & \text{🏪} \vdash \Theta, z : \text{🎂}, z' : \text{🎂} \\
\vdots & \vdots & \vdots
\end{array}
$$
$$
(\nu x x')((\star x[z].\text{👦} \parallel \star x[z'].\text{👩}) \parallel \star x'(w).\star x'(w').\text{🏪}) \vdash \Gamma, \Delta, \Theta
$$

The constructs $\star x[y].P$ and $\star x(y).P$ introduce a single client or server action, respectively—hence, channels of type $!_1$ and $?_1$. However, when we cut, we want to cut on both interactions simultaneously. We need rules for the *contraction* of shared channel names.

4.1. Clients and Pooling.

A client pool represents a number of independent processes, each wanting to interact with the same server. Examples of such a pool include Ami and Boé from our example, customers for online stores in general, and any number of processes which interact with a single, centralised server.

We introduce two new rules: one to construct clients, and one to pool them together. The first rule, $(!_1)$, interacts over a channel as a client. It does this by receiving a channel y over a *shared* channel x. The channel y is the channel across which the actual interaction will eventually take place. The second rule, Cont!, allows us to contract shared channel names with the same type. When used together with H-Mix, this allows us to pool clients together.

$$
\frac{P \vdash \Gamma, y : A}{\star x[y].P \vdash \Gamma, x : !_1A} \; (!_1) \qquad \frac{P \vdash \mathcal{G} \parallel \Gamma, x : !_mA \parallel \Delta, x' : !_nA}{P\{x/x'\} \vdash \mathcal{G} \parallel \Gamma, \Delta, x : !_{m+n}A} \; \text{Cont!}
$$

Using these rules, we can derive the left-hand side of our proof by marking Ami and Boé as clients, and pooling them together.

$$
\cfrac{
\cfrac{
\cfrac{\text{👦} \vdash \Gamma, z : \text{🎂}^\perp}{\star x[z].\text{👦} \vdash \Gamma, z : !_1\text{🎂}^\perp} \; (!_1)
\qquad
\cfrac{\text{👩} \vdash \Delta, z' : \text{🎂}^\perp}{\star x'[z'].\text{👩} \vdash \Delta, x' : !_1\text{🎂}^\perp} \; (!_1)
}{(\star x[z].\text{👦} \parallel \star x'[z'].\text{👩}) \vdash \Gamma, x : !_1\text{🎂}^\perp \parallel \Delta, x' : !_1\text{🎂}^\perp} \; \text{H-Mix}
}{(\star x[z].\text{👦} \parallel \star x[z'].\text{👩}) \vdash \Gamma, \Delta, x : !_2\text{🎂}^\perp} \; \text{Cont!}
$$

4.2. Servers and Sequencing.

Dual to a pool of n clients in parallel is a server with n actions in sequence. Our interpretation of a server is a process which offers some number of interdependent interactions of the same type. Examples include the store from our example, which gives out slices of cake and disappointment, online stores in general, and any central server which interacts with some number of client processes.

We introduce two new rules to construct servers. The first rule, $(?_1)$, marks a interaction over some channel as a server interaction. It does this by sending a channel y over a *shared* channel x. The channel y is the channel across which the actual interaction will take place. The second rule, Cont?, allows us to merge two (possibly interleaved) sequences of server interactions. This allows us to construct a server which has multiple interactions of the same type, across the same shared channel.

$$\frac{P \vdash \Gamma, y : A}{\star x(y).P \vdash \Gamma, x : ?_1 A}\ (?_1) \qquad \frac{P \vdash \mathcal{G} \parallel \Gamma, x : ?_m A, x' : ?_n A}{P\{x/x'\} \vdash \mathcal{G} \parallel \Gamma, x : ?_{m+n} A}\ \text{CONT}_?$$

Using these rules, we can derive the right-hand side of our proof, by marking each of the store's interactions as server interactions, and then contracting them.

$$\cfrac{\cfrac{\cfrac{\text{🏪} \vdash \Theta, w : \text{🎂}, w' : \text{🎂}}{\star y'(w').\text{🏪} \vdash \Theta, z : \text{🎂}, y' : ?_1 \text{🎂}}\ (?_1)}{\star y(w).\star y'(w').\text{🏪} \vdash \Theta, y : ?_1 \text{🎂}, y' : ?_1 \text{🎂}}\ (?_1)}{\star y(w).\star x(w').\text{🏪} \vdash \Theta, y : ?_2 \text{🎂}}\ \text{CONT}_?$$

Thus, we complete the typing derivation of our example.

Definition 4.3 (Typing judgements). We extend theorem 3.7 as follows:

$$\frac{P \vdash \Gamma, y : A}{\star x[y].P \vdash \Gamma, x : !_1 A}\ (!_1) \qquad \frac{P \vdash \Gamma, y : A}{\star x(y).P \vdash \Gamma, x : ?_1 A}\ (?_1)$$

$$\frac{P \vdash \mathcal{G} \parallel \Gamma, x : !_m A \parallel \Delta, x' : !_n A}{P\{x/x'\} \vdash \mathcal{G} \parallel \Gamma, \Delta, x : !_{m+n} A}\ \text{CONT}_! \qquad \frac{P \vdash \mathcal{G} \parallel \Gamma, x : ?_m A, x' : ?_n A}{P\{x/x'\} \vdash \mathcal{G} \parallel \Gamma, x : ?_{m+n} A}\ \text{CONT}_?$$

4.3. **Running Clients and Servers.** Finally, we need to extend the reduction rules to allow for the reduction of client and server processes. The reduction rule we add is a variant of E-SEND.

Definition 4.4 (Reduction). We extend theorem 3.8 as follows:

E-REQUEST $(\nu x x')((\star x[y].P \parallel \star x'(y').Q) \parallel R) \implies (\nu x x')((\nu y y')(P \parallel Q) \parallel R)$

The difference between E-REQUEST and E-SEND is that the former allows reduction to happen in the presence of an unrelated process R, which is passed along unchanged. This is necessary, as there may be other clients waiting to interact with the server on the shared channel x, which cannot be moved out of scope of the name restriction (νx). When there is no unrelated process R, *i.e.*, when there is only a single client, we can rewrite by SC-PARNIL before and after applying E-REQUEST.

So where does the non-determinism in HCP_{ND} come from? Let us say we have a term of the following form:

$$(\nu x x')((\star x[y_1].P_1 \parallel \cdots \mid \star x[y_n].P_n) \parallel \star x'(y'_1).\ldots.\star x'(y'_n).Q)$$

As parallel composition is commutative and associative, we can rewrite this term to pair any client in the pool with the server before applying E-REQUEST. Thus, like in the π-calculus, the non-determinism is introduced by the structural congruence.

Does this mean that, for an arbitrary client pool P in $(\nu x y)(P \parallel \star y(w).Q)$, every client in that pool is competing for the server interaction on x? Not necessarily, as some portion of the clients can be blocked on an external communication. For instance, in the term below, clients $\star x[z_{n+1}].P_{n+1} \ldots \star x[z_m].P_m$ are blocked on a communication on the external channel a:

$$\begin{aligned}(\nu x x')((&\ (\star x[y_1].P_1 \parallel \cdots \mid \star x[y_n].P_n) \\ &\mid a().(\star x[y_{n+1}].P_{n+1} \parallel \cdots \mid \star x[y_m].P_m)\ ) \\ &\mid \star x'(y'_1).\ldots.\star x'(y'_m).Q\ )\end{aligned}$$

If we reduce this term, then only the clients $\star x[z_1].P_1 \ldots \star x[z_n].P_n$ will be assigned server interactions, and we end up with the following canonical form:

$$(\nu x x')(\; a().(\star x[y_{n+1}].P_{n+1} \parallel \cdots \mid \star x[y_m].P_m)$$
$$\mid \star x'(y'_{n+1}). \ldots . \star x'(y'_m).Q \;)$$

This matches our intuition and the behaviour of the π-calculus. For instance, we can now encode our example, where Ami and Boé both send a request for cake to the store, and the store sends back either a cake or nothing:

$$(\nu x x') \begin{pmatrix} \star x[x].x(y).\text{🧑} \parallel \\ \star x[x].x(z).\text{👩} \parallel \\ \star x'(x'_1).x'_1\langle\text{🍰}\rangle.\star x'(x'_2).x'_2\langle\text{🙌}\rangle.\text{🏪} \end{pmatrix} \Longrightarrow^\star \begin{array}{c} (\text{🧑}\{\text{🍰}/y\} \parallel \text{👩}\{\text{🙌}/z\} \parallel \text{🏪}) \\ \text{or} \\ (\text{🧑}\{\text{🙌}/y\} \parallel \text{👩}\{\text{🍰}/z\} \parallel \text{🏪}) \end{array}$$

The encoding presented above is slightly more complex than necessary: after the store receives a request as x'_1, it could simply perform the cake interaction over that channel, and similarly for x'_2. However, we include these actions for clarity.

Alternative syntax. If we choose to reuse the terms $x[y].P$ and $x(y).P$ for shared channels, we could replace E-Send with E-Request, using the latter rule for both cases.

4.4. **Metatheory.** HCP$_{\text{ND}}$ enjoys subject reduction, termination, and progress.

Lemma 4.5 (Preservation for ≡). *If $P \equiv Q$ and $P \vdash \mathcal{G}$, then $Q \vdash \mathcal{G}$.*

Proof. By induction on the derivation of $P \equiv Q$. □

Theorem 4.6 (Preservation). *If $P \vdash \mathcal{G}$ and $P \Longrightarrow Q$, then $Q \vdash \mathcal{G}$.*

Proof. By induction on the derivation of $P \Longrightarrow Q$. □

Definition 4.7 (Actions). A process P acts on x whenever x is free in the outermost term constructor of P, *e.g.*, $\star x(y).P$ acts on x but not on y, and $x \leftrightarrow y$ acts on both x and y. A process P is an action if it acts on some channel x.

Definition 4.8 (Canonical forms). A process P is in canonical form if
$$P \equiv (\nu x_1 x'_1) \ldots (\nu x_n x'_n)(P_1 \mid \cdots \mid P_{n+m+1}),$$
such that: no process P_i is a cut or a mix; no process P_i is a link acting on a bound channel x_i or x'_i; and no two processes P_i and P_j are acting on dual endpoints x_i and x'_i of the same channel.

Lemma 4.9. *If a well-typed process P is in canonical form, then it is blocked on an external communication, i.e., $P \equiv (\nu x_1 x'_1) \ldots (\nu x_n x'_n)(P_1 \mid \cdots \mid P_{n+m+1})$ such that at least one process P_i acts on a free name.*

Proof. We have $P \equiv (\nu x_1 x'_1) \ldots (\nu x_n x'_n)(P_1 \parallel \ldots \parallel P_{n+m+1})$, such that no P_i is a cut or a link acting on a bound channel, and no two processes P_i and P_j are acting on the same bound channel with dual actions. The prefix of cuts and mixes introduces n channels. Each application of cut requires an application of mix, so the prefix introduces $n+m+1$ processes. Each application of Cont! requires an application of mix, so there are at most m clients acting on the same bound channel. Therefore, at least *one* of the processes P_i must be acting on a free channel, i.e., blocked on an external communication. □

Theorem 4.10 (Progress). *If $P \vdash \mathcal{G}$, then either P is in canonical form, or there exists a process Q such that $P \Longrightarrow Q$.*

Proof. We consider the maximum prefix of cuts and mixes of P such that

$$P \equiv (\nu x_1 x_1') \ldots (\nu x_n x_n')(P_1 \parallel \ldots \parallel P_{n+m+1}),$$

and no P_i is a cut. If any process P_i is a link, we reduce by $(\leftrightarrow)$. If any two processes P_i and P_j are acting dual endpoints x_i and x_i' of the same channel, we rewrite by $\equiv$ and reduce by the appropriate β-rule. Otherwise, P is in canonical form.   □

Theorem 4.11 (Termination). *If $P \vdash \mathcal{G}$, then there are no infinite $\Longrightarrow$-reduction sequences.*

Proof. Every reduction reduces a single cut to zero, one or two cuts. However, each of these cuts is smaller, measured in the size of the cut formula. Furthermore, each instance of the structural congruence preserves the size of the cut. Therefore, there cannot be an infinite $\Longrightarrow$-reduction sequence.   □

4.5. $\mathbf{HCP_{ND}}$ and Non-deterministic Local Choice.

In section 2, we discussed the non-deterministic local choice operator, which is used in several extensions of πDILL and CP [ALM16, Cai14, CP17]. This operator is admissible in $\mathrm{HCP_{ND}}$. We can derive the non-deterministic choice $P + Q$ by constructing the following term:

$$(\nu x x')((\;\star x[y].y \vartriangleleft \mathtt{inl}.y[].0$$
$$|\;\star x[z].z \vartriangleleft \mathtt{inr}.z[].0\;)$$
$$|\;\star x'(y').\star x'(z').y' \vartriangleright$$
$$\{\mathtt{inl}:\;(\nu w w')(z' \vartriangleright \{\mathtt{inl}: z'().w[].0; \mathtt{inr}: z'().w[].0\} \parallel w'().P)$$
$$;\mathtt{inr}:\;(\nu w w')(z' \vartriangleright \{\mathtt{inl}: z'().w[].0; \mathtt{inr}: z'().w[].0\} \parallel w'().Q)\;\})$$

This term is a cut between two processes.

- On the left-hand side, we have a pool of two processes, $\star x[y].y \vartriangleleft \mathtt{inl}.y[].0$ and $\star x[z].z \vartriangleleft \mathtt{inr}.z[].0$. Each makes a choice: the first sends $\mathtt{inl}$, and the second sends $\mathtt{inr}$.
- On the right-hand side, we have a server with both P and Q. This server has two channels on which a choice is offered, y' and z'. The choice on y' selects between P and Q. The choice on z' does not affect the outcome of the process at all. Instead, it is discarded.

When these clients and the server are put together, the choices offered by the server will be non-deterministically lined up with the clients which make choices, and either P or Q will run.

While there is a certain amount of overhead involved in this encoding, it scales linearly in terms of the number of processes. The reverse—encoding the non-determinism present in $\mathrm{HCP_{ND}}$ using non-deterministic local choice—scales exponentially, see, *e.g.*, the examples in section 2.

5. Cuts with Leftovers

So far, our account of a non-determinism in client/server interactions only allows for interactions between equal numbers of clients and server interactions. A natural question is whether or not we can deal with the scenario in which there are more client than server interactions or vice versa, *i.e.*, whether or not the following rules are derivable:

$$\frac{\vdash \Gamma, !_{n+m}A \qquad \vdash \Delta, ?_nA^\perp}{\vdash \Gamma, \Delta, !_mA} \qquad \frac{\vdash \Gamma, !_nA \qquad \vdash \Delta, ?_{n+m}A^\perp}{\vdash \Gamma, \Delta, ?_mA^\perp}$$

These rules are derivable using a link. For instance, we can derive the rule for the case in which there are more clients than servers as follows:

$$\frac{P \vdash \Gamma, x\!:\!!_{n+m}A \qquad \dfrac{\dfrac{Q \vdash \Delta, x'\!:\!?_nA^\perp \qquad x''\!\leftrightarrow\!w \vdash x''\!:\!?_mA^\perp, w\!:\!!_mA}{(Q \parallel x''\!\leftrightarrow\!w) \vdash \Delta, x'\!:\!?_nA^\perp \parallel x''\!:\!?_mA^\perp, w\!:\!!_mA}\;\text{H-Mix}}{(Q \parallel x'\!\leftrightarrow\!w) \vdash \Delta, x'\!:\!?_{n+m}A^\perp, w\!:\!!_mA}\;\text{Cont}_!}{\dfrac{(P \parallel (Q \parallel x'\!\leftrightarrow\!w)) \vdash \Gamma, x\!:\!!_{n+m}A \parallel \Delta, x'\!:\!?_{n+m}A^\perp, w\!:\!!_mA}{(\nu xx')(P \parallel (Q \parallel x'\!\leftrightarrow\!w)) \vdash \Gamma, \Delta, w\!:\!!_mA}\;\text{Cut}}\;\text{H-Mix}}$$

6. Relation to Manifest Sharing

In section 2, we mentioned related work which extends πDILL and CP with non-deterministic local choice [ALM16, Cai14, CP17], and contrasted these approaches with ours. In this section, we will contrast our work with the more recent work on manifest sharing [BP17].

Manifest sharing extends the session-typed language SILL with two connectives, $\uparrow_L^S A$ and $\downarrow_L^S A$, which represent the places in a protocol where a shared resource is aquired and released, respectively. In the resulting language, SILL_S, we can define a type for, *e.g.*, shared queues (using the notation for types introduced in this paper):

$$\text{queue } A ::= \uparrow_L^S (\, A^\perp \otimes \downarrow_L^S(\text{queue } A)\,) \,\&\, (\,(A \oplus \perp) \otimes \downarrow_L^S(\text{queue } A)\,)$$

The type queue A types a shared channel which, after we aqcuire exclusive access, gives us the choice between enqueuing a value ($A^\perp$) and releasing the queue, or dequeuing a value if there is any ($A \oplus \perp$) and releasing the queue.

The language SILL_S is much more expressive than HCP_{ND}, as it has support for both shared channels and recursion. In fact, Balzer, Pfenning, and Toninho [BPT18] show that SILL_S is expressive enough to embed the untyped asynchronous π-calculus. This expressiveness comes with a cost, as SILL_S processes are not guaranteed to be deadlock free, though recent work addresses this issue [BTP19].

Despite the difference in expressiveness, there are some similarities between HCP_{ND} and SILL_S. In the former, shared channels represent (length-indexed) streams of interactions of the same type. In the latter, it is necessary for type preservation that shared channels are always released at the same type at which they were acquired, meaning that shared channels also represent (possibly infinite) streams of interactions of the same type. In fact, in HCP_{ND}, the type for queues (with n interactions) can be written as $!_n(A^\perp \,\&\, (A \oplus \perp))$.

One key difference between HCP_{ND} and SILL_S is that in SILL_S a server must finish interacting with one client before interacting with another, whereas in HCP_{ND} the server may interact with multiple clients simultaneously.

7. Discussion and Future Work

We presented HCP_{ND}, an extension of HCP which permits non-deterministic communication without losing the strong connection to logic. We gave proofs for preservation, progress, and termination for the term reduction system of HCP_{ND}. We showed that we can define non-deterministic local choice in HCP_{ND}.

Our formalism so far has only captured servers that provide for a fixed number of clients. More realistically, we would want to define servers that provide for arbitrary numbers of clients. This poses two problems: how would we define arbitrarily-interacting stateful processes, and how would we extend the typing discipline of $\mathrm{HCP_{ND}}$ to account for them without losing its static guarantees.

One approach to defining server processes would be to combine $\mathrm{HCP_{ND}}$ with structural recursion and corecursion, following the μCP extension of Lindley and Morris [LM16]. Their approach can express processes which produce streams of A channels. Such a process would expose a channel with the co-recursive type $\nu X.A \bindnasrepma (1 \oplus X)$. Given such a process, it is possible to produce a channel of type $A \bindnasrepma A \bindnasrepma \cdots \bindnasrepma A$ for any number of As, allowing us to satisfy the type $?_n A$ for an arbitrary n.

We would also need to extend the typing discipline to capture arbitrary use of shared channels. One approach would be to introduce resource variables and quantification. Following this approach, in addition to having types $?_n A$ and $!_n A$ for concrete n, we would also have types $?_x A$ and $!_x A$ for resource variables x. These variables would be introduced by quantifiers $\forall x A$ and $\exists x A$. Defining terms corresponding to $\forall x A$, and its relationship with structured recursion, presents an interesting area of further work.

Our account of HCP did not include the exponentials $?A$ and $!A$. The type $!A$ denotes arbitrarily many independent instances of A, while the type $?A$ denotes a concrete (if unspecified) number of potentially-dependent instances of A. Existing interpretations of linear logic as session types have taken $!A$ to denote A-servers, while $?A$ denotes A-clients. However, the analogy is imperfect: while we expect servers to provide arbitrarily many instances of their behaviour, we also expect those instances to be interdependent.

With quantification over resource variables, we can give precise accounts of both CP's exponentials and idealised servers and clients. CP exponentials could be embedded into this framework using the definitions $!A ::= \forall n !_n A$ and $?A ::= \exists n ?_n A$. We would also have types that precisely matched our intuitions for server and client behavior: an A server is of type $\forall n ?_n A$, as it serves an unbounded number of requests with the requests being interdependent, while a collection of A clients is of type $\exists n !_n A$, as we have a specific number of clients with each client being independent.

References

[ALM16] Robert Atkey, Sam Lindley, and J. Garrett Morris. Conflation confers concurrency. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Lecture Notes in Computer Science, 2016.

[Bor98] Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theoretical Computer Science*, 195(2):205–226, 3 1998.

[BP17] Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–29, August 2017.

[BPT18] Stephanie Balzer, Frank Pfenning, and Bernardo Toninho. A Universal Session Type for Untyped Asynchronous Communication. In Sven Schewe and Lijun Zhang, editors, *29th International Conference on Concurrency Theory (CONCUR 2018)*, volume 118 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:18, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[BTP19] Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest deadlock-freedom for shared session types. In *Programming Languages and Systems*, pages 611–639. Springer International Publishing, 2019.

[Cai14]   Luís Caires. Types and logic, concurrency and non-determinism. In Martin Abadi, Philippa
           Gardner, Andy Gordon, and Radu Mardare, editors, *Essays for the Luca Cardelli Fest.* Microsoft
           Research, 9 2014.
[CP10]   Luís Caires and Frank Pfenning. *Session Types as Intuitionistic Linear Propositions*, pages 222–236.
           Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
[CP17]   Luís Caires and Jorge A. Pérez. 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.* Springer, 4 2017.
[GSS92]   Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: A modular approach
           to polynomial-time computability. *Theor. Comput. Sci.*, 97(1):1–66, April 1992.
[Hon93]   Kohei Honda. Types for dyadic interaction. In *CONCUR'93*, pages 509–523. Springer Nature,
           1993.
[KMP18]   Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Taking linear logic apart. In *Workshop on
           Linearity & TLLA at FloC'18*, July 2018.
[KMP19]   Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Better late than never: A fully-abstract
           semantics for classical processes. *PACMPL*, 3(POPL), January 2019.
[LM15]   Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *Programming
           Languages and Systems*, pages 560–584. Springer Nature, 2015.
[LM16]   Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. In
           Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming,
           ICFP 2016, pages 434–447, New York, NY, USA, 2016. ACM.
[MPW92]   Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Information
           and Computation*, 100(1):41–77, September 1992.
[San96]   Davide Sangiorgi. π-calculus, internal mobility, and agent-passing calculi. *Theoretical Computer
           Science*, 167(1-2):235–274, 1996.
[Vas12]   Vasco T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012.
[Wad12]   Philip Wadler. Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN International
           Conference on Functional Programming*, ICFP '12, pages 273–286, New York, NY, USA, 2012.
           ACM.