# Conflation Confers Concurrency

Robert Atkey[2], Sam Lindley[1], and J. Garrett Morris[1]

[1] The University of Edinburgh
{Sam.Lindley,Garrett.Morris}@ed.ac.uk
[2] University of Strathclyde
Robert.Atkey@strath.ac.uk

**Abstract.** Session types provide a static guarantee that concurrent programs respect communication protocols. Recent work has explored a correspondence between proof rules and cut reduction in linear logic and typing and evaluation of process calculi. This paper considers two approaches to extend logically-founded process calculi. First, we consider extensions of the process calculus to more closely resemble $\pi$-calculus. Second, inspired by denotational models of process calculi, we consider conflating dual types. Most interestingly, we observe that these approaches coincide: conflating the multiplicatives ($\otimes$ and $\parr$) allows processes to share multiple channels; conflating the additives ($\oplus$ and $\&$) provides nondeterminism; and conflating the exponentials (! and ?) yields access points, a rendezvous mechanism for initiating session typed communication. Access points are particularly expressive: for example, they are sufficient to encode concurrent state and general recursion.

## 1 Introduction

The Curry-Howard correspondence, formulated by Howard [1980] and building on ideas of Curry [1934] and Tait [1965], observes a remarkable correspondence between (propositional) intuitionistic logic and the $\lambda$-calculus. The correspondence identifies logical propositions with $\lambda$-calculus types, proofs with well-typed terms, and cut elimination with reduction. The correspondence is fruitful from the perspectives of both logic and programming languages [Wadler, 2015].

Recent work, initiated by Caires and Pfenning [2010], attempts a similar identification between linear logic and process calculi. In this case, propositions of (intuitionistic) linear logic are identified with session types, a mechanism for typing interacting processes originally proposed by Honda [1993]. Proofs of those propositions are identified with $\pi$-calculus processes, and cut elimination in linear logic with $\pi$-calculus reduction. However, this identification is not as satisfying as that for intuitionistic logic and the $\lambda$-calculus. In particular, numerous features of the $\pi$-calculus are excluded by the resulting session-typing discipline, and there is not yet an approach to restore them. Philip Wadler posed the following question at an invited talk by Frank Pfenning at the TLDI Workshop in January 2012 [paraphrased]:

> Simply-typed $\lambda$-calculus has a propositions as types interpretation as intuitionistic logic, but has limited expressiveness. By adding a fix point operator it becomes Turing-complete. Similarly, Caires and Pfenning's calculus has a propositions as types interpretation as intuitionistic linear logic. Is there a feature, analogous to the fix point operator, that we can add to Caires and Pfenning's calculus in order to recover the full power of $\pi$-calculus?

Wadler has since reposed the same question, but with respect to his classical variant of Caires and Pfenning's system (Wadler [2014]).

This paper describes an approach to more expressive, logically founded process calculi, inspired by Wadler's question. We begin by considering extensions of Wadler's CP calculus that increase its expressiveness—at the cost of properties such as deadlock freedom—while retaining session fidelity (well-typed communication). To do so, we explore two approaches. On the one hand, we directly investigate the inclusion of $\pi$-calculus terms excluded by CP's type system, bringing CP more in line with the term calculi in most existing presentations of session types. Doing so requires the addition of new typing rules, and we consider their interpretation as proof rules and their logical consequences. On the other hand, inspired by the semantics of the $\pi$-calculus described by Abramsky et al. [1996], we attempt to conflate the various dual types. Our primary contribution is the observation that these disparate approaches converge: the new proof rules allow us to show bi-implications between dual types, while assuming such bi-implications make the new logical rules derivable.

The paper proceeds as follows. We begin with a short introduction to session types and their connection to linear logic and its semantics (§2). We recall Wadler's CP calculus (§3). We then describe the conflation of the various dual types and their consequences (§4). Finally, we conclude with a discussion of future work and open questions (§5).

## 2 Background

Unlike the propositions of intuitionistic logic, linear logic propositions are finite resources—the assumptions of a proof must each be used exactly once in the course of the proof. When he introduced linear logic, Girard [1987] suggested that it might be suited to reasoning about parallelism. Abramsky [1992] and Bellin and Scott [1994] give embeddings of linear logic proofs in $\pi$-calculus, and show that cut reduction is simulated by $\pi$-calculus reduction. Their work is not intended to provide a type system for $\pi$-calculus: there are many processes which are not the image of some proof.

Session types were originally introduced by Honda [1993] as a type system for $\pi$-calculus-like communicating processes. The key constructors of session types include input and output, characterising the exchange of data, and internal and external choice, characterising branching evaluation. Honda's typing discipline assures *session fidelity*, meaning that at each synchronisation the communicating

processes agree on the types of values exchanged. His system is extended to $\pi$-calculus-like processes by Takeuchi et al. [1994] and Honda et al. [1998]. Session typing relies on a substructural type system to assure session fidelity; however, Honda did not relate his types to the propositions of linear logic, and he relies on a self-dual type for closed channels. Kobayashi et al. [1996] give a linear, typed polyadic $\pi$-calculus, in which each channel must be used exactly once, and show how it can be used to encode serial uses, as in session types; they do not address choice or nondeterminism.

Recent work by Caires and Pfenning [2010] and Wadler [2014], among others, has developed a propositions-as-types correspondence between session typing and linear logic. Session types are interpreted via the connectives of linear logic—for example, the tensor product $A \otimes B$ characterises processes that output values of type $A$, and then proceed as $B$. Caires and Pfenning [2010] interpret the proof rules for intuitionistic linear logic as a type system for the $\pi$-calculus; they then show that the reduction steps for well-typed $\pi$-calculus terms correspond to the cut elimination rules for intuitionistic linear logic. Wadler [2014] adapts their approach to classical linear logic, emphasising the role of duality in typing; in his system, the semantics of terms is given directly by the cut elimination rules. As a consequence of their logical connections, both systems enjoy deadlock freedom, termination, and a lack of nondeterministic behaviour; however, both systems also impose additional limitations on the underlying process calculus, and differ in some ways from traditional presentations of session types.

Abramsky et al. [1996] propose an alternative approach to typing communicating processes. Their approach is based on a denotational model for processes, called interaction categories. Their canonical interaction category, called **SCons** is sufficient to interpret linear logic; unusually, the interpretations of the dual connectives are conflated. **SCons** is quite expressive: it can faithfully interpret all of Milner's synchronous CCS calculus. However, we are not aware of any work extending interaction categories to interpret channel-passing calculi, such as $\pi$-calculus or the calculi of Caires and Pfenning and Wadler.

There have been several type systems for deadlock-free processes not derived from linear logic, including those of Kobayashi [2006], Padovani [2014], and Giachino et al. [2014]. These systems all include composition of processes sharing multiple channels, but at the cost of additional type-system complexity. Dardha and Pérez [2015] compare Kobayashi's approach with the linear-logic based approaches of Caires, Pfenning, and Wadler. They observe that the linear logic-based approaches coincide with that of Kobayashi when restricted to one shared channel between processes. They also give a rewriting procedure from Kobayashi-typable processes with multiple shared channels to processes with one shared channel, and show an operational correspondence between the original and rewritten processes.

Some of our results above are reminiscent of results obtained in category-theoretic settings closely related to linear logic. Fiore [2007] shows that conflation of products and coproducts into biproducts is equivalent to the existence of a monoidal structure on morphisms; we will attempt a similar conflation (§4.3)

to give a logical interpretation of nondeterminism. Compact closed categories provide a model of classical linear logic in which $\otimes$ and $\parr$ are identified; we will consider a similar conflation as well (§4.2). However, we do not yet understand the exact relationship between compact closed categories, and our system with the multicut rule. In compact closed categories it is known that finite products are automatically biproducts Houston [2008], and it would be interesting to see how his proof translates to a logical setting.

Finally, there have been several attempts to relate $\pi$-calculus to proof nets, one approach to the semantics of linear logic. Honda and Laurent [2010] demonstrate a two-way correspondence between proof nets for polarised linear logic and a typed $\pi$-calculus previous presented by Honda et al. [2004]. The type system used for the $\pi$-calculus in this work is rather restrictive, however: it ensures that all processes are deterministic and effectively sequential, removing much of the expressivity of the $\pi$-calculus. Ehrhard and Laurent [2010] give a translation from finitary $\pi$-calculus (i.e., $\pi$-calculus without replication) into differential interaction nets. Differential interaction nets are an untyped formalism for representing computation, based on proof nets for differential linear logic. Differential linear logic is an extension of linear logic that, amongst other features adds a form of nondeterminism. It is this nondeterminism that Ehrhard and Laurent use to model the $\pi$-calculus. However, Mazza [2015] argues forcefully that Ehrhard and Laurent's translation incorrectly models the nondeterminism present in the $\pi$-calculus. In short, Mazza shows that differential proof nets can only model "localised" nondeterminism—nondeterminism that can be resolved via a local coin flip—and not the global nondeterminism that arises when two processes "race" to communicate with another process. Mazza makes the following provocative statement:

> However, although linear logic has kept providing, even in recent times, useful tools for ensuring properties of process algebras, especially via type systems (Kobayashi et al., 1999; Yoshida et al., 2004; Caires and Pfenning, 2010; Honda and Laurent, 2010), all further investigations have failed to bring any deep logical insight into concurrency theory, in the sense that no concurrent primitive has found a convincing counterpart in linear logic, or anything even remotely resembling the perfect correspondence between functional languages and intuitionistic logic. In our opinion, we must simply accept that linear logic is not the right framework for carrying out Abramsky's "proofs as processes" program (which, in fact, more than 20 years after its inception has yet to see a satisfactory completion).

We will show that, while not entirely answering Mazza's critique, our identifications provide some logical justification for $\pi$-calculus-like features.

## 3 Classical Linear Logic and the Process Calculus CP

We begin by reviewing Wadler's CP calculus [Wadler, 2014], its typing, and its semantics. For simplicity, we restrict ourselves to the propositional fragment of CP, omitting second-order existential and universal quantification.

### 3.1 Types and Duality

The types of the CP calculus are built from the multiplicative, additive, and exponential propositional connectives of Girard's classical linear logic (CLL).

$$\text{Types } A, B ::= A \otimes B \mid A \parr B \mid 1 \mid \bot \mid A \oplus B \mid A \,\&\, B \mid 0 \mid \top \mid !A \mid ?A$$

Wadler's contribution with the CP calculus was to give the logical connectives of CLL an explicit reading in terms of session types. As is standard with session typing, CP types denote the types of channel end points. The connectives come in dual pairs, indicating complementary obligations for each end point of a channel. The tensor connective $A \otimes B$ means output $A$ and then behave like $B$; and dually, par $A \parr B$ means input $A$ and then behave like $B$. The unit of $\otimes$ is 1, empty output; dually, the unit of $\parr$ is $\bot$, empty input. Internal choice $A \oplus B$ means make a choice between $A$ and $B$; dually, external choice $A \,\&\, B$ means accept a choice of $A$ or $B$. Replication $!A$ means produce an arbitrary number of copies of $A$; dually, query $?A$ means consume a copy of $A$.

The dual relationships between $\otimes$ and $\parr$, 1 and $\bot$, $\oplus$ and $\&$, and $!$ and $?$ are formalised in the duality operation $-^\bot$, which takes each type to its dual.

$$\begin{array}{lll} (A \otimes B)^\bot = A^\bot \parr B^\bot & (A \oplus B)^\bot = A^\bot \,\&\, B^\bot & \\ (A \parr B)^\bot = A^\bot \otimes B^\bot & (A \,\&\, B)^\bot = A^\bot \oplus B^\bot & (!A)^\bot = ?(A^\bot) \\ 1^\bot = \bot & \top^\bot = 0 & (?A)^\bot = !(A^\bot) \\ \bot^\bot = 1 & 0^\bot = \top & \end{array}$$

*Example: Sending and receiving bits.* CP is a rather low-level calculus. The multiplicative fragment ($\otimes$ and $\parr$) handles only sending and receiving of channels, on which data may be subsequently transmitted. The only actual data that may be transmitted between processes are single bits, which take the form of a choice between a pair of sessions. Transmission of a single bit is represented by internal choice between two empty outputs.

$$\mathit{Bool} = 1 \oplus 1$$

Dually, receiving a single bit is represented by an external choice between two empty inputs.

$$\mathit{Bool}^\bot = \bot \,\&\, \bot$$

The other propositional connectives of CLL can now be used to build more complex specifications. For example, we can write down the type of a server

that offers a choice of a binary operation on booleans (single bits) and a unary operation on booleans, arbitrarily many times.

$$\mathit{Server} = !((\mathit{Bool}^{\perp} \parr \mathit{Bool}^{\perp} \parr \mathit{Bool} \otimes 1) \,\&\, (\mathit{Bool}^{\perp} \parr \mathit{Bool} \otimes 1))$$

The outer ! indicates that an implementation of this type is obliged to offer the server as many times as a client requires. The inner part of the type is a client-choice, indicated by the external choice (&), between the two operations. The left-hand choice $\mathit{Bool}^{\perp} \parr \mathit{Bool}^{\perp} \parr \mathit{Bool} \otimes 1$ can be read as "the server must input two booleans, output a boolean, and then signal the end of the session". The right-hand choice is similar, but only specifies the input of a single boolean input.

A compatible client type is obtained by taking the dual of the *Server* type.

$$\mathit{Client} = \mathit{Server}^{\perp} = ?((\mathit{Bool} \otimes \mathit{Bool} \otimes \mathit{Bool}^{\perp} \parr \perp) \oplus (\mathit{Bool} \otimes \mathit{Bool}^{\perp} \parr \perp))$$

Dually to the server's obligations, the outer ? indicates that the client may make as many uses of this session as it desires. The inner part of the type is an internal choice ($\oplus$), indicating that the implementation of this type must make a choice between the two services. The two choices then describe the same pattern as the server, but from the point of view of the client. The left-hand choice $\mathit{Bool} \otimes \mathit{Bool} \otimes \mathit{Bool}^{\perp} \parr \perp$ can be read as "the client must output two booleans, input a boolean, and then signal the end of the session". The right-hand choice is similar, but only specifies the output of a single boolean.

### 3.2 Terms and Typing

The terms of CP are given by the following grammar.

$$\begin{aligned}\text{Terms } P, Q ::= {} & x \leftrightarrow y \mid \nu y\,(P \mid Q) \mid x(y).P \mid x[y].(P \mid Q) \mid !x(y).P \mid ?x[y].P \\ & \mid x[in_i].P \mid \mathsf{case}\ x.\{P; Q\} \mid x().P \mid x[].0 \mid \mathsf{case}\ x.\{\}\end{aligned}$$

Figure 1 gives the typing rules of CP. The typing judgement is of the form $P \vdash \Gamma$, where $P$ is a CP process term, and $\Gamma$ is a channel typing environment. In rule (Bang), $?\Gamma$ denotes a context $\Gamma$ in which all types are of the form $?A$ for some type $A$. Note that CP's typing rules implicitly rebind identifiers: for example, in the hypothesis of the rule for $\parr$, $x$ identifies a proof of $B$, while in the conclusion it identifies a proof of $A \parr B$.

CP includes two rules that are logically derivable: the axiom rule, which is interpreted as channel forwarding, and the cut rule, which is interpreted as process composition. Two of CP's terms differ from standard $\pi$-calculus terms. The first is composition—rather than having distinct name restriction and composition operators, CP provides one combined operator. This syntactically captures the restriction that composed processes must share exactly one channel. The second is output: the CP term $x[y].(P \mid Q)$ includes output, composition, and name restriction (the name $y$ designates a new channel, bound in $P$). Finally, note that CP includes only replicated input, not arbitrary replicated processes.

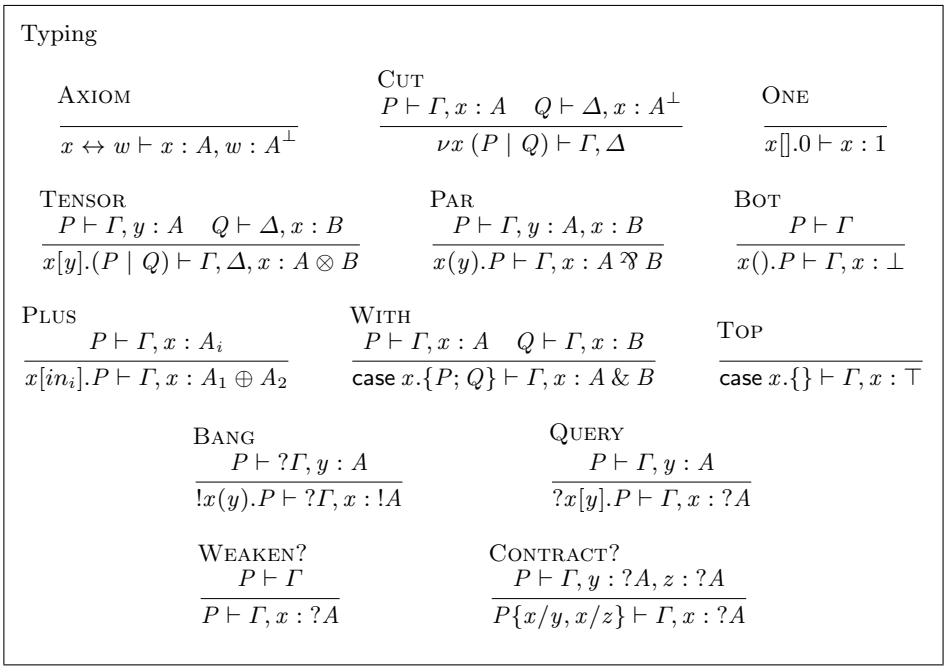


**Fig. 1:** CP Typing

*A simpler send.* The CP rule Tensor is appealing because if one erases the terms it is exactly the classical linear logic rule for tensor. However, this correspondence comes at a price. Operationally, the process $x[y].(P \mid Q)$ does three things: it introduces a fresh variable $y$, it sends $y$ to a freshly spawned process $P$, and in parallel it continues as process $Q$. Fortunately, we can straightforwardly define the operation that sends a free variable along a channel as syntactic sugar (Lindley and Morris [2015] discuss this in more detail).

$$x\langle y\rangle.P \stackrel{\text{def}}{=} x[z].(y \leftrightarrow z \mid P)$$

*Example: Sending and receiving bits.* As mentioned above, CP is quite low-level, so we use some syntactic sugar for sending and receiving bits.

$$\begin{aligned} x[0].P &\stackrel{\text{def}}{=} x[y].(y[in_1].y[].0 \mid P) \\ x[1].P &\stackrel{\text{def}}{=} x[y].(y[in_2].y[].0 \mid P) \\ \mathsf{case}\ p.\{0 \mapsto P; 1 \mapsto Q\} &\stackrel{\text{def}}{=} \mathsf{case}\ p.\{p().P; p().Q\} \end{aligned}$$

Let us define a logic server process $P \vdash x : \mathit{Server}$, in which the binary operation is "and", and the unary operation is "not".

$$\begin{aligned} P = {} & !x(y).\mathsf{case}\ y.\{y(p).y(q). \\ & \qquad \mathsf{case}\ p.\{0 \mapsto \mathsf{case}\ q.\{0 \mapsto y[0].y[].0; 1 \mapsto y[0].y[].0\}; \\ & \qquad\qquad 1 \mapsto \mathsf{case}\ q.\{0 \mapsto y[0].y[].0; 1 \mapsto y[1].y[].0\}\}; \\ & \quad y(p). \\ & \qquad \mathsf{case}\ p.\{0 \mapsto y[1].y[].0; 1 \mapsto y[0].y[].0\}\} \end{aligned}$$

This process operates over replicated channel $x$, accepting a channel $y$. A client process communicating along the other end of $x$ will begin by choosing between the "and" and "not" operations. If "and" is requested, then two bits ($p$ and $q$) are received along $y$, and their logical conjunction is sent back along $y$. If "not" is requested, then a single bit ($p$) is received along $y$, and its logical negation is sent back along $y$.

We now define a client process $Q$ that uses $P$ to compute "not ($p$ and $q$)", using the result to choose between two processes $P_0$ and $P_1$ as continuations.

$$\begin{aligned} Q = {} & ?x[y].y[\mathit{in}_1].y\langle p\rangle.y\langle q\rangle.y(z).y(). \\ & ?x[y].y[\mathit{in}_2].y\langle z\rangle.y(r).y(). \\ & \mathsf{case}\ r.\{0 \mapsto P_0; 1 \mapsto P_1\} \end{aligned}$$

The process $Q$ connects to $P$ twice using channel $x$, selecting the "and" operation, and then the "not" operation. We have that $Q \vdash p : \mathit{Bool}^{\perp}, q : \mathit{Bool}^{\perp}, x : \mathit{Client}, \Delta$ whenever $P_0 \vdash \Delta$ and $P_1 \vdash \Delta$.

### 3.3 Semantics via Cut Reduction

The semantics of CP terms are given by cut reduction, as shown in Figure 2. We write $\mathit{fv}(P)$ for the free names of process $P$. Terms are identified up to structural congruence $\equiv$ (as name restriction and composition are combined into one form, composition is not always associative). We write $\longrightarrow_{\mathrm{C}}$ for the cut reduction relation and $\longrightarrow_{\mathrm{CC}}$ for the commuting conversion relation. We denote sequential composition of relations by juxtaposition. We write $R^*$ for the transitive reflexive closure and $R^?$ for the reflexive closure of relation $R$.

The majority of the cut reduction rules correspond closely to synchronous reductions in $\pi$-calculus—for example, the reduction of & against $\oplus$ corresponds to the synchronisation of an internal and external choice. The rule for reduction of $⅋$ against $\otimes$ is more complex than synchronisation of input and output in $\pi$-calculus, as it must also manipulate the implicit name restriction and parallel composition in the output term. We deviate slightly from Wadler's presentation in that substitution and duplication and discarding of replicated processes happens in the structural congruence rules for weakening and contraction and not as part of the rule for dereliction. These choices ensure that reduction only concerns interactions between dual prefixes and other non-intensional behaviour is handled by the structural congruence.

Structural congruence

$$
\begin{aligned}
x \leftrightarrow w &\equiv w \leftrightarrow x \\
\nu x\,(w \leftrightarrow x \mid P) &\equiv P[w/x] \\
\nu x\,(P \mid Q) &\equiv \nu x\,(Q \mid P) \\
\nu x\,(P \mid \nu y\,(Q \mid R)) &\equiv \nu y\,(\nu x\,(P \mid Q) \mid R), \quad \text{if } x \notin fv(R) \\
\nu x\,(!x(y).P \mid Q) &\equiv Q, \qquad x \notin fv(Q) \\
\nu x\,(!x(y).P \mid Q\{x/z\}) &\equiv \nu x\,(!x(y).P \mid \nu z\,(!z(y).P \mid Q))
\end{aligned}
$$

Primary cut reduction rules

$$
\begin{aligned}
\nu x\,(x[].0 \mid x().P) &\longrightarrow_{\mathrm{C}} P \\
\nu x\,(x[y].(P \mid Q) \mid x(y).R) &\longrightarrow_{\mathrm{C}} \nu x\,(Q \mid \nu y\,(P \mid R)) \\
\nu x\,(x[in_i].P \mid \mathsf{case}\ x.\{Q_1; Q_2\}) &\longrightarrow_{\mathrm{C}} \nu x\,(P \mid Q_i) \\
\nu x\,(!x(y).P \mid ?x[y].Q) &\longrightarrow_{\mathrm{C}} \nu y\,(P \mid Q), \quad \text{if } x \notin fv(Q)
\end{aligned}
$$

$$
\frac{P \longrightarrow_{\mathrm{C}} P'}{\nu x\,(P \mid Q) \longrightarrow_{\mathrm{C}} \nu x\,(P' \mid Q)}
$$

Commuting conversions

$$
\begin{aligned}
\nu z\,(x[y].(P \mid Q) \mid R) &\longrightarrow_{\mathrm{CC}} x[y].(\nu z\,(P \mid R) \mid Q), \qquad \text{if } z \notin fv(Q) \\
\nu z\,(x[y].(P \mid Q) \mid R) &\longrightarrow_{\mathrm{CC}} x[y].(P \mid \nu z\,(Q \mid R)), \qquad \text{if } z \notin fv(P) \\
\nu z\,(x(y).P \mid Q) &\longrightarrow_{\mathrm{CC}} x(y).\nu z\,(P \mid Q) \\
\nu z\,(x().P \mid Q) &\longrightarrow_{\mathrm{CC}} x().\nu z\,(P \mid Q) \\
\nu z\,(x[in_i].P \mid Q) &\longrightarrow_{\mathrm{CC}} x[in_i].\nu z\,(P \mid Q) \\
\nu z\,(\mathsf{case}\ x.\{P; Q\} \mid R) &\longrightarrow_{\mathrm{CC}} \mathsf{case}\ x.\{\nu z\,(P \mid R); \nu z\,(Q \mid R)\} \\
\nu z\,(\mathsf{case}\ x.\{\} \mid Q) &\longrightarrow_{\mathrm{CC}} \mathsf{case}\ x.\{\} \\
\nu z\,(!x(y).P \mid Q) &\longrightarrow_{\mathrm{CC}} !x(y).\nu z\,(P \mid Q) \\
\nu z\,(?x[y].P \mid Q) &\longrightarrow_{\mathrm{CC}} ?x[y].\nu z\,(P \mid Q)
\end{aligned}
$$

**Fig. 2:** CP Congruences and Cut Reduction

**Theorem 1 (Preservation).** *The relations* $\equiv$, $\longrightarrow_C$, *and* $\longrightarrow_{CC}$ *preserve well-typing: if* $P \vdash \Gamma$ *then* $P \equiv Q$ *implies* $Q \vdash \Gamma$, $P \longrightarrow_C Q$ *implies* $Q \vdash \Gamma$, *and* $P \longrightarrow_{CC} Q$ *implies* $Q \vdash \Gamma$.

Just as cut elimination in logic ensures that any proof may be transformed into an equivalent cut-free proof, the reduction rules of CP transform any term into a term blocked on external communication—that is to say, if $P \vdash \Gamma$, then $P \, (\equiv \longrightarrow_{\mathrm{C}})^* (\equiv \longrightarrow_{\mathrm{CC}})^? \, P'$ where $P' \neq \nu x \, (Q \mid Q')$ for any $x, Q, Q'$. The optional final commuting conversion plays a crucial role in this transformation, moving any remaining internal communication after some external communication (precluding deadlock). Note, however, that commuting conversions do not correspond to computational steps (i.e., any reduction rule in $\pi$-calculus).

*Example: Sending and receiving bits.* Recall that, under the assumption of two processes $P_0 \vdash \Delta$ and $P_1 \vdash \Delta$, we have $P \vdash x : \mathit{Server}$ and $Q \vdash p : \mathit{Bool}^{\perp}, q : \mathit{Bool}^{\perp}, x : \mathit{Client}, \Delta$. In order to connect the client and server, we bind $x$ to the replicated channel they communicate along: $PQ = \nu x \, (P \mid Q)$. We can set the values of the bits $p$ and $q$ by placing further processes in parallel with $PQ$, which allows reduction. For instance,

$$\nu p \, (p[1] \mid \nu q \, (q[1] \mid PQ)) \longrightarrow_{\mathrm{C}}^* P_0$$

and:

$$\nu p \, (p[0] \mid \nu q \, (q[1] \mid PQ)) \longrightarrow_{\mathrm{C}}^* P_1$$

*Properties of cut reduction.* Unsurprisingly, being a term calculus for classical linear logic, CP is well-behaved. CP cut reduction is terminating, CP does not admit deadlocks, and CP is deterministic.

**Theorem 2 (Termination).** *The relation* $\longrightarrow_C$ *is strongly-normalising modulo* $\equiv$*: if* $P \vdash \Gamma$, *then there are no infinite* $\equiv \longrightarrow_C$ *reduction sequences starting from* $P$.

**Theorem 3 (Deadlock-freedom).** *If* $P \vdash \Gamma$, *then there exist* $P', Q$ *such that* $P \, (\equiv \longrightarrow_C)^* \, P'$ *and* $P \, (\equiv \longrightarrow_{CC})^? \, Q$, *and* $Q$ *is not a cut.*

**Theorem 4 (Determinism).** *The relation* $\longrightarrow_C$ *is confluent modulo* $\equiv$*: if* $P \vdash \Gamma$, $P \, (\equiv \longrightarrow_C)^* \, Q_1$ *and* $P \, (\equiv \longrightarrow_C)^* \, Q_2$, *then there exist* $R_1, R_2$ *such that* $Q_1 \, (\equiv \longrightarrow_C)^* \, R_1$, $Q_2 \, (\equiv \longrightarrow_C)^* \, R_2$, *and* $R_1 \equiv R_2$.

All of these theorems follow from well known results about classical linear logic.

## 4 Conflating Duals

In light of the results above about CP's semantics, it would seem that CP is not a particularly expressive calculus. In particular, nondeterminism is frequently seen as a defining characteristic of concurrency, and the guaranteed deterministic

behaviour of CP (Theorem 4) would appear to indicate that it is not possible to express much interesting behaviour in CP.

One property of CP that appears to be essential for its behavioural properties is its strict adherence to duality. Each connective has an accompanying dual, and the fact that communicating processes must match dual connectives precisely means that "nothing goes wrong", in the sense of non-termination, deadlock or nondeterministic behaviour. Observing that this precise matching makes CP a relatively inexpressive calculus, we now systematically investigate relaxation of the strict duality of CP to see whether or not it yields greater expressivity by conflating each of the dual pairs of connectives in turn.

*How to conflate duals.* There are choices over exactly how to conflate duals in a proof theory. In a standard session-typed calculus, for instance, one literally replaces 1 and $\bot$ with a single type (usually called $\mathsf{end}$). However, in order to keep our modified calculus conservative with respect to the existing one, we initially take a different approach following Wadler [2014], who considers extensions of CP in which certain maps between duals are derivable. Logically, we can say that we have conflated propositions (session types) $A$ and $B$ if there exist processes $P_{AB}$ and $P_{BA}$ such that $P_{AB} \vdash x : A \multimap B$ and $P_{BA} \vdash x : B \multimap A$ (where $A \multimap B = A^{\bot} \parr B$), which amounts to giving back-and-forth translations between $A$ and $B$. In this paper, we do not necessarily require that these translations be mutually inverse in any way, though we usually expect this to be the case. Our general pattern will be to add some feature (parallelism, multi-channel cut, nondeterminism, access points) and then observe that this is logically equivalent to conflation of a dual pair.

As indicated above, we define $A \multimap B \stackrel{\text{def}}{=} A^{\bot} \parr B$. Furthermore, $\parr$ is invertible

$$\dfrac{P \vdash z : A \parr B \qquad \dfrac{\dfrac{}{w \leftrightarrow x \vdash w : A^{\bot}, x : A} \quad \dfrac{}{y \leftrightarrow z \vdash y : B, z : B^{\bot}}}{\vdash z : A^{\bot} \otimes B^{\bot}, x : A, y : B}}{\nu z\,(P \mid z[w].(w \leftrightarrow x \mid y \leftrightarrow z)) \vdash x : A, y : B}$$

so without loss of generality we shall seek $P_{AB}$ and $P_{BA}$ such that $P_{AB} \vdash x : A^{\bot}, y : B$ and $P_{BA} \vdash x : B^{\bot}, y : A$ and we shall systematically treat a proof $P_{AB} \vdash x : A^{\bot}, y : B$ as a proof of $A \multimap B$.

### 4.1 Concurrency without Communication (1 and $\bot$)

The first pair of connectives we conflate are 1 and $\bot$. Conflation of this pair corresponds to the addition of communication-free concurrency and inactive processes to CP, via the Mix and 0-Mix rules.

The Mix rule allows processes $P$ and $Q$ to be composed in parallel $P \mid Q$ without creating a communication channel between them.

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

The unit of Mix is the inactive process 0.

$$\frac{}{0 \vdash}\ \text{0-Mix}$$

Using Mix we can prove $\bot \multimap 1$.

$$\frac{\dfrac{}{x[].0 \vdash x : 1} \quad \dfrac{}{y[].0 \vdash y : 1}}{x[].0 \mid y[].0 \vdash x : 1, y : 1}$$

Conversely, if we already have a proof $P_{\bot 1}$ of $\bot \multimap 1$, then we can derive a proof of the Mix rule.

$$\frac{\dfrac{P \vdash \Gamma}{x().P \vdash \Gamma, x : \bot} \quad \dfrac{\dfrac{Q \vdash \Delta}{y().Q \vdash \Delta, y : \bot} \quad \dfrac{}{P_{\bot 1} \vdash x : 1, y : 1}}{\nu y\,(Q \mid P_{\bot 1}) \vdash \Delta, x : 1}}{\nu x\,(P \mid \nu y\,(Q \mid P_{\bot 1})) \vdash \Gamma, \Delta}$$

Thus, Mix and $\bot \multimap 1$ are logically equivalent and we choose to define a process $P_{\bot 1}$ in terms of Mix.

$$P_{\bot 1} \stackrel{\text{def}}{=} x[].0 \mid y[].0$$

In the other direction, using 0-Mix we can prove $1 \multimap \bot$.

$$\frac{\dfrac{\dfrac{}{0 \vdash}}{y().0 \vdash y : \bot}}{x().y().0 \vdash x : \bot, y : \bot}$$

Conversely, if we already have a proof $P_{1\bot}$ of $1 \multimap \bot$, then we can derive a proof of the 0-Mix rule.

$$\frac{\dfrac{\dfrac{}{P_{1\bot} \vdash x : \bot, y : \bot} \quad \dfrac{}{y[].0 \vdash y : 1}}{\nu y\,(P_{1\bot} \mid y[].0) \vdash x : \bot} \quad \dfrac{}{x[].0 \vdash x : 1}}{\nu x\,(\nu y\,(P_{1\bot} \mid y[].0) \mid x[].0) \vdash}$$

Thus, 0-Mix and $1 \multimap \bot$ are logically equivalent and we choose to define the process $P_{1\bot}$ in terms of 0-mix.

$$P_{1\bot} \stackrel{\text{def}}{=} x().y().0$$

*Semantics.* To account for Mix and 0-Mix, we extend the structural congruence with the following rules.

$$\begin{aligned} P \mid 0 &\equiv P \\ P \mid Q &\equiv Q \mid P \\ P \mid (Q \mid R) &\equiv (P \mid Q) \mid R \\ \nu x\,(P \mid (Q \mid R)) &\equiv (\nu x\,(P \mid Q)) \mid R, \quad \text{if } x \notin fv(R) \end{aligned}$$

The only cut reduction rule we add allows reduction under a Mix.

$$\frac{P \longrightarrow_{\mathrm{C}} P'}{P \mid Q \longrightarrow_{\mathrm{C}} P' \mid Q}$$

For each standard commuting conversion involving a cut, there is a corresponding one where the cut is replaced by a Mix.

$$\begin{aligned}
x[y].(P \mid Q) \mid R &\longrightarrow_{\mathrm{CC}} x[y].((P \mid R) \mid Q) \\
x[y].(P \mid Q) \mid R &\longrightarrow_{\mathrm{CC}} x[y].(P \mid (Q \mid R)) \\
(x(y).P) \mid Q &\longrightarrow_{\mathrm{CC}} x(y).(P \mid Q) \\
(x().P) \mid Q &\longrightarrow_{\mathrm{CC}} x().(P \mid Q) \\
(x[in_i].P) \mid Q &\longrightarrow_{\mathrm{CC}} x[in_i].(P \mid Q) \\
\mathsf{case}\ x.\{P; Q\} \mid R &\longrightarrow_{\mathrm{CC}} \mathsf{case}\ x.\{P \mid R; Q \mid R\} \\
\mathsf{case}\ x.\{\} \mid Q &\longrightarrow_{\mathrm{CC}} \mathsf{case}\ x.\{\}
\end{aligned}$$

Adding Mix and 0-Mix does not disturb any of the usual meta-theoretic properties of CP: termination, deadlock-freedom, and determinism all still hold. Therefore, the conflation of 1 and $\bot$ does not greatly alter the properties of CP, though it is does weaken the separation between $\otimes$ and $\parr$, as we shall see in the next section.

### 4.2 Concurrency with Multiple Channels ($\otimes$ and $\parr$)

The second pair of connectives we consider for conflation are $\otimes$ and $\parr$. This conflation is logically equivalent to the addition of a multi-cut rule that enables communication over several channels at the same time, generalising the single channel of the standard Cut rule, and the zero channels of the Mix rule. As we shall see, the introduction of this rule results in the possibility of deadlock.

Conflation of the units 1 and $\bot$ of $\otimes$ and $\parr$ via Mix already yields a translation from the former to the latter.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x \leftrightarrow y \vdash x : A^{\bot}, y : A} \quad \dfrac{}{w \leftrightarrow z \vdash w : B^{\bot}, z : B}}{x \leftrightarrow y \mid w \leftrightarrow z \vdash x : A^{\bot}, w : B^{\bot}, y : A, z : B}}{y(z).(x \leftrightarrow y \mid w \leftrightarrow z) \vdash x : A^{\bot} \parr B^{\bot}, y : A, z : B}}{x(w).y(z).(x \leftrightarrow y \mid w \leftrightarrow z) \vdash x : A^{\bot} \parr B^{\bot}, y : A \parr B}$$

Conversely, if we already have a proof $P_{\otimes\parr}$ of $A \otimes B \multimap A \parr B$, then we can prove $\vdash 1, 1$, and hence the Mix rule

$$\dfrac{\dfrac{\dfrac{}{P_{\otimes\parr} \vdash x : 1 \parr 1, y : \bot \parr \bot} \quad \dfrac{\dfrac{}{\vdash x : 1} \quad \dfrac{}{\vdash y : 1}}{y[x].(x[].0 \mid y[].0) \vdash y : 1 \otimes 1}}{\nu y\ (P_{\otimes\parr} \mid y[x].(x[].0 \mid y[].0)) \vdash x : 1 \parr 1}}{\nu x\ (\nu y\ (P_{\otimes\parr} \mid y[x].(x[].0 \mid y[].0)) \mid x[w].w \leftrightarrow z \mid x \leftrightarrow y) \vdash y : 1, z : 1}$$

where we make use of the invertibility of $\parr$.

Whereas Mix is a variation on cut in which no channels communicate between the two processes, another natural variation is to allow two channels to communicate simultaneously between two processes.

$$\textsc{BiCut}\ \frac{P \vdash \Gamma, x : A^\perp, y : B^\perp \quad Q \vdash \Delta, x : A, y : B}{\nu xy\ (P \mid Q) \vdash \Gamma, \Delta}$$

In general one might consider $n$ channels communicating across a cut so the Mix rule is the nullary case of such a multicut.

Using BiCut we can show that $A \parr B \multimap A \otimes B$.

$$\frac{\dfrac{\dfrac{}{y : A^\perp, w : A} \quad \dfrac{}{x : B^\perp, z : B}}{x : A^\perp \otimes B^\perp, w : A, z : B} \quad \dfrac{\dfrac{}{x : A, w : A^\perp} \quad \dfrac{}{y : B, z : B^\perp}}{y : A \otimes B, w : A^\perp, z : B^\perp}}{\begin{array}{r}\nu wz\ (x[y].(w \leftrightarrow y \mid x \leftrightarrow z) \\ \mid y[x].(w \leftrightarrow x \mid y \leftrightarrow z))\end{array} \vdash x : A^\perp \otimes B^\perp, y : A \otimes B}$$

Conversely, if we already have a proof $P_{\parr\otimes}$ of $A \parr B \multimap A \otimes B$, then we can derive a proof of the BiCut rule.

$$\frac{\dfrac{P \vdash \Gamma, x : A^\perp, y : B^\perp}{y[x].P \vdash \Gamma, y : A^\perp \otimes B^\perp} \quad \dfrac{Q \vdash \Delta, x : A, y : B}{y(x).Q \vdash \Delta, y : A \parr B}}{\nu y\ (y[x].P \mid y(x).Q) \vdash \Gamma, \Delta}$$

where we write $y[x].P$ as syntactic sugar for $\nu z\ (z(x).P\{z/y\} \mid P_{\parr\otimes})$.

$$\frac{\dfrac{\dfrac{P \vdash \Gamma, x : A, y : B}{P\{z/y\} \vdash \Gamma, x : A, z : B}}{z(y).P\{z/y\} \vdash \Gamma, z : A \parr B} \quad \dfrac{}{P_{\parr\otimes} \vdash z : (A \parr B)^\perp, y : A \otimes B}}{\nu z\ (z(x).P\{z/y\} \mid P_{\parr\otimes}) \vdash \Gamma, y : A \otimes B}$$

Thus, BiCut and $A \parr B \multimap A \otimes B$ are logically equivalent and we choose to define the process $P_{\parr\otimes}$ in terms of BiCut.

$$P_{\parr\otimes} = \nu wz\ (x[y].(w \leftrightarrow y \mid x \leftrightarrow z) \mid y[x].(w \leftrightarrow x \mid y \leftrightarrow z))$$

We can also derive 0-Mix using BiCut and Axiom.

$$0 \stackrel{\text{def}}{=} \nu xy\ (x \leftrightarrow y \mid x \leftrightarrow y) \vdash$$

As 0 does not have any observable behaviour in any context and neither does this term, we can perfectly well take this to be our definition of 0 when working with variants of CP that include BiCut.

*From bicut to multicut.* A natural way to try to define cut reduction in the presence of BiCut is to lift each of the existing cut rules to bicut rules in which the inactive name remains in place. Unfortunately, this does not work for the rule reducing the interaction between $\otimes$ introduction and $\parr$ introduction (i.e. sending and receiving a fresh channel).

$$\nu zx\,(x[y].(P \mid Q) \mid x(y).R) \longrightarrow_{\mathrm{C}} \nu zy\,(P \mid \nu x\,(Q \mid R))$$

The problem is that $z$ may be bound in $Q$, so moving $Q$ across the $z$ cut may require contradictory types for $z$. One can attempt to work around the issue by analysing whether $z$ occurs in $P$ or $Q$. But if $z$ has type $?A$ for some $A$ then it may occur in both. To avoid the problem we take a more uniform approach, introducing a multicut construct that generalises Mix, BiCut, and Cut.

$$\textsc{MultiCut}\quad \frac{P \vdash \Gamma, x_1 : A_1^{\perp}, \ldots, x_n : A_n^{\perp} \qquad Q \vdash \Delta, x_1 : A_1, \ldots, x_n : A_n}{\nu x_1, \ldots, x_n\,(P \mid Q) \vdash \Gamma, \Delta}$$

Now we can construct a send/receive rule that does not move any of the subterms across a cut. (We shall delegate all such cut-shuffling to the structural congruence.)

$$\nu\vec{x}x\,(x[y].(P \mid Q) \mid x(y).R) \longrightarrow_{\mathrm{C}} \nu\vec{x}xy\,((P \mid Q) \mid R)$$

Extending the pattern for BiCut, one can encode $n$-cut, for $n > 2$ in terms of $P_{\parr\otimes}$ (and hence BiCut).

$$\nu x_1\ \ldots\ x_n\,(P \mid Q) \stackrel{\text{def}}{=} \nu x_n\,(x_n[x_1].\cdots x[x_{n-1}].P \mid x_n(x_1).\cdots x_n(x_{n-1}).Q)$$

This shows us that BiCut is logically equivalent to MultiCut, but it does not help with defining the semantics of BiCut, which we specify using MultiCut anyway.

*Semantics.* The structural congruence in the presence of MultiCut is as follows.

$$\begin{aligned}
\nu\vec{x}xy\vec{y}\,(P \mid Q) &\equiv \nu\vec{x}yx\vec{y}\,(P \mid Q)\\
x \leftrightarrow w &\equiv w \leftrightarrow x\\
\nu\vec{x}x\,(w \leftrightarrow x \mid P) &\equiv P[w/x], \quad \text{if } w \notin \vec{x}\\
\nu\vec{x}\,(P \mid Q) &\equiv \nu\vec{x}\,(Q \mid P)\\
\nu\vec{x}\vec{y}\,(P \mid \nu\vec{z}\,(Q \mid R)) &\equiv \nu\vec{y}\vec{z}\,(\nu\vec{x}\,(P \mid Q) \mid R),\\
&\qquad \text{if } \vec{x} \notin fv(R) \text{ and } \vec{z} \notin fv(P)\\
\nu\vec{x}x\,(!x(y).P \mid Q) &\equiv Q, \qquad \text{if } x \notin fv(Q)\\
\nu\vec{x}x\,(!x(y).P \mid Q\{x/z\}) &\equiv \nu\vec{x}xz\,((!x(y).P \mid !z(y).P) \mid Q)
\end{aligned}$$

The first rule allows reordering of bound variables. The remaining rules correspond to the original structural rules, taking into account the possibility of additional cut variables. The third rule is constrained such that a substitution

can only be performed if the substituted variable is not bound by the cut. The fifth rule allows us to focus on any pair of processes while moving the cut variables between the multicuts appropriately. The final rule takes advantage of Mix to avoid the problem we already encountered with adapting the original send/receive rule for use with BiCut.

The primary cut rules are as follows.

$$\begin{aligned}
\nu\vec{x}\,(x[].0 \mid x().P) &\longrightarrow_{\mathrm{C}} P \\
\nu\vec{x}x\,(x[y].(P \mid Q) \mid x(y).R) &\longrightarrow_{\mathrm{C}} \nu\vec{x}xy\,((P \mid Q) \mid R) \\
\nu\vec{x}x\,(x[in_i].P \mid \mathsf{case}\ x.\{Q_1; Q_2\}) &\longrightarrow_{\mathrm{C}} \nu\vec{x}x\,(P \mid Q_i) \\
\nu\vec{x}x\,(!x(y).P \mid ?x[y].Q) &\longrightarrow_{\mathrm{C}} \nu\vec{x}y\,(P \mid Q), \quad \text{if } x \notin fv(Q)
\end{aligned}$$

$$\frac{P \longrightarrow_{\mathrm{C}} P'}{\nu\vec{x}\,(P \mid Q) \longrightarrow_{\mathrm{C}} \nu\vec{x}\,(P' \mid Q)}$$

The send/receive rule uses mix and extends the existing multicut. The other rules are straightforward generalisations of the original ones.

Adding MultiCut introduces the possibility of deadlock. Termination (but not cut-elimination) and determinism are preserved.

### 4.3 Nondeterminism (0 and $\top$ / $\oplus$ and &)

The previous two sections dealt with the conflation of the multiplicative connectives, $1/\bot$ and $\otimes/\parr$. Conflation of the additive connectives $0/\top$, and $\oplus/\&$ yields a calculus that has *local* nondeterminism: processes can be defined as the nondeterministic combination of two processes, or as processes that may fail.

As in $\pi$-calculus, we can easily extend CP with a construct to nondeterministically choose between two processes.

$$\textsc{Choose}\ \frac{P \vdash \Gamma \quad Q \vdash \Gamma}{P + Q \vdash \Gamma}$$

The unit of nondeterminism $\mathsf{fail}$ makes no choices.

$$\textsc{Fail}\ \frac{}{\mathsf{fail} \vdash \Gamma}$$

Binary nondeterminism does not change the properties we can prove, and indeed the CHOOSE rule is derivable in two ways.

$$\dfrac{\dfrac{\dfrac{P \vdash \Gamma}{x().P \vdash \Gamma, x : \bot} \quad \dfrac{Q \vdash \Gamma}{x().Q \vdash \Gamma, x : \bot}}{\mathsf{case}\ x.\{x().P; x().Q\} \vdash \Gamma, x : \bot \,\&\, \bot} \quad \dfrac{\dfrac{}{x[].0 \vdash x : 1}}{x[in_1].x[].0 \vdash x : 1 \oplus 1}}{\nu x\ (\mathsf{case}\ x.\{x().P; x().Q\} \mid x[in_1].x[].0) \vdash \Gamma}$$

$$\dfrac{\dfrac{\dfrac{P \vdash \Gamma}{x().P \vdash \Gamma, x : \bot} \quad \dfrac{Q \vdash \Gamma}{x().Q \vdash \Gamma, x : \bot}}{\mathsf{case}\ x.\{x().P; x().Q\} \vdash \Gamma, x : \bot \,\&\, \bot} \quad \dfrac{\dfrac{}{x[].0 \vdash x : 1}}{x[in_2].x[].0 \vdash x : 1 \oplus 1}}{\nu x\ (\mathsf{case}\ x.\{x().P; x().Q\} \mid x[in_2].x[].0) \vdash \Gamma}$$

The proofs are not cut-free, and they both fail to capture the semantics of nondeterminism: the first always reduces to $P$ and the second to $Q$.

We do not need to add any features to CP in order to prove that $0 \multimap \top$. Logically 0 represents falsehood, and thus implies everything including $\top$. However, there are two distinct cut-free proofs of $0 \multimap \top$.

$$\dfrac{}{\mathsf{case}\ x.\{\} \vdash x : \top, y : \top} \qquad \dfrac{}{\mathsf{case}\ y.\{\} \vdash x : \top, y : \top}$$

We can combine these using nondeterminism.

$$\mathsf{case}\ x.\{\} + \mathsf{case}\ y.\{\} \vdash x : \top, y : \top$$

Similarly, there are two distinct cut-free proofs of $A \,\&\, B \multimap A \oplus B$ in plain CP.

$$\dfrac{\dfrac{\dfrac{}{x \leftrightarrow y \vdash A^\bot, A}}{y[in_1].x \leftrightarrow y \vdash x : A^\bot, y : A \oplus B}}{x[in_1].y[in_1].x \leftrightarrow y \vdash x : A^\bot \oplus B^\bot, y : A \oplus B}$$

$$\dfrac{\dfrac{\dfrac{}{x \leftrightarrow y \vdash x : B^\bot, y : B}}{y[in_2].x \leftrightarrow y \vdash x : B^\bot, y : A \oplus B}}{x[in_2].y[in_2].x \leftrightarrow y \vdash x : A^\bot \oplus B^\bot, y : A \oplus B}$$

Again, we can combine these using nondeterminism.

$$(x[in_1].y[in_1].x \leftrightarrow y) + (x[in_2].y[in_2].x \leftrightarrow y) \vdash x : A^\bot \oplus B^\bot, y : A \oplus B$$

Reading CP as a logic, the FAIL rule has obvious logical problems: it allows us to prove anything! However, that is not to say that it does not have a meaningful dynamic semantics in terms of cut reduction. Failure immediately yields a proof of $\top \multimap 0$.

$$\dfrac{}{\mathsf{fail} \vdash x : 0, y : 0}$$

Conversely, if we already have a proof $P_{\top 0}$ of $\top \multimap 0$, then we can derive a proof of the FAIL rule.

$$\dfrac{\dfrac{}{\mathsf{case}\ x.\{\} \vdash x : \top} \quad \dfrac{P_{\top 0} \vdash x : 0, y : 0 \quad \overline{\mathsf{case}\ y.\{\} \vdash y : \top, \Gamma}}{\nu y\ (P_{\top 0} \mid \mathsf{case}\ y.\{\}) \vdash x : 0, \Gamma}}{\nu x\ (\mathsf{case}\ x.\{\} \mid \nu y\ (P_{\top 0} \mid \mathsf{case}\ y.\{\})) \vdash \Gamma}$$

Thus, FAIL and $\top \multimap 0$ are logically equivalent and we can define $P_{\top 0}$ in terms of FAIL.

$$P_{\top 0} \stackrel{\text{def}}{=} \mathsf{fail}$$

Of course, $A \oplus B \multimap A \,\&\, B$ can be proved with $\mathsf{fail}$. We could simply use $\mathsf{fail}$ to prove this immediately, but this does not have the right dynamic semantics in terms of nondeterminism. A more satisfying proof is given by the following judgement.

$$\mathsf{case}\ x.\{\mathsf{case}\ y.\{x \leftrightarrow y; \mathsf{fail}\}; \mathsf{case}\ y.\{\mathsf{fail}; x \leftrightarrow y\}\} \vdash \Gamma$$

This judgement captures the idea that reduction can succeed if the two channels make compatible choices. In the other direction, we can prove $\vdash 0$ (and hence $\vdash \Gamma$ for any $\Gamma$) from $P_{\oplus\&} \vdash A \oplus B \multimap A \,\&\, B$.

$$\dfrac{\dfrac{\dfrac{\overline{x[].0 \vdash x{:}1}}{P \vdash x{:}1 \oplus 0} \quad \overline{P_{\oplus\&} \vdash x{:}\bot \,\&\, \top, y{:}1 \,\&\, 0}}{\nu x\ (P \mid P_{\oplus\&}) \vdash y : 1 \,\&\, 0} \quad \dfrac{\overline{\mathsf{case}\ y.\{\} \vdash y{:}\top, z{:}0}}{Q \vdash y{:}\bot \oplus \top, z{:}0}}{\nu y\ (\nu x\ (x[in_1].x[].0 \mid P_{\oplus\&}) \mid y[in_2].\mathsf{case}\ y.\{\}) \vdash z{:}0}$$

where $\qquad P = x[in_1].x[].0 \qquad Q = y[in_2].\mathsf{case}\ y.\{\}$

Thus, FAIL and $A \oplus B \multimap A \,\&\, B$ are logically equivalent and we choose to define the semantics of $P_{\oplus\&}$ in terms of FAIL.

$$P_{\oplus\&} \stackrel{\text{def}}{=} \mathsf{case}\ x.\{\mathsf{case}\ y.\{x \leftrightarrow y; \mathsf{fail}\}; \mathsf{case}\ y.\{\mathsf{fail}; x \leftrightarrow y\}\}$$

*Bicut and fail.* It turns out that FAIL is also derivable from BICUT. If we have BICUT then we can define FAIL as follows.

$$\mathsf{fail} \stackrel{\text{def}}{=} \nu xy\ (x \leftrightarrow y \mid \mathsf{case}\ x.\{\})$$

*Semantics.* Following $\pi$-calculus, we add two reduction rules.

$$P + Q \longrightarrow_{\mathrm{C}} P$$
$$P + Q \longrightarrow_{\mathrm{C}} Q$$

Nondeterminism (without BiCut) does not disturb cut-elimination. Termination and deadlock-freedom are preserved. Of course, determinism is lost. However, the determinism we have added is *local*, in the sense that while an individual process may nondeterministically evolve, this nondeterminism does not arise from two separate processes racing for some shared resource. If we are to capture interesting concurrent behaviour then it seems natural to allow for racy communication. We will see in the next section how to introduce races.

### 4.4 Access Points (! and ?)

Access points provide a dynamic match-making service for initiating session communication. An access point $a$ of type $\star A$ allows an arbitrary number of channels of type $A$ to nondeterministically accept requests from an arbitrary number of channels of type $A^\perp$. Each channel of type $A$ is nondeterministically paired up with a matching channel of type $A^\perp$. If at any point the numbers of $A$ channels and $A^\perp$ channels differ, then any unmatched channels block until a matching channel becomes available (which may never happen).

It is not clear to us if it is possible to conflate ! and ? in quite the same way as we did for the other type constructors. Nevertheless, we can achieve a similar effect by a slightly different route, whereby we replace ! and ? with access points.

The rules for *accepting* and *requesting* a connection through an access point replace the Bang and Query rules.

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

The two differences from the Bang and Query rules are that: in each rule $x$ is bound in $P$, allowing the access point to be reused in the continuation; and there are no restrictions on $\Gamma$ in Accept. The weakening and contraction capabilities are extended to propositions of the form $!A$.

$$\textsc{Weaken!}\ \frac{P \vdash \Gamma}{P \vdash \Gamma, x : !A} \qquad \textsc{Contract!}\ \frac{P \vdash \Gamma, y : !A, z : !A}{P\{x/y, x/z\} \vdash \Gamma, x : !A}$$

The original Bang and Query rules are derivable from the above rules:

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

Conversely, if for all $A$ we have proofs of $!A \multimap ?A$ and $?A \multimap !A$, then we can straightforwardly derive Accept, Request, Weaken!, and Contract!.

Access points liberate name restriction from parallel composition. We will use the following syntactic sugar.

$$\begin{aligned} 0 &\stackrel{\text{def}}{=} \nu a\,(\star a[x].x[].0 \mid \star a[x].x[].0) \\ (\nu a)P &\stackrel{\text{def}}{=} \nu a\,(P \mid 0) \\ (\nu a_1 \ldots a_n)P &\stackrel{\text{def}}{=} (\nu a)\ldots(\nu a_n)P \end{aligned}$$

In addition to the changes to the typing rules, we also add a type equation

$$\star A = !A = ?A^{\perp}$$

capturing the idea that access points play the role of both ! and ?. This equation immediately implies that $!A \multimap ?A^{\perp}$ and $?A \multimap !A^{\perp}$, which does not quite fit with our previous pattern. However, if it is the case that $A$ and all of its subformulas are self-dual then we can show that $!A \multimap ?A$ and $?A \multimap !A$ are admissible, as we might expect. As it turns out, we can encode Mix, MultiCut, and Choice using access points, so indeed these properties are admissible and all types are self-dual.

$$\begin{aligned} P \mid Q &\stackrel{\text{def}}{=} \nu a\,(P \mid Q) \\ \nu x_1 \ldots x_n\,(P \mid Q) &\stackrel{\text{def}}{=} (\nu a_1 \ldots a_n)(\star a_1(x_1).\cdots\star a_n(x_n).P \mid \star a_n[x_1].\cdots\star a_n[x_n].Q) \end{aligned}$$

$$\begin{aligned} \mathsf{fail} &\stackrel{\text{def}}{=} \nu x\,(\mathsf{case}\ x.\{\} \mid (\nu a)\star a(y).x \leftrightarrow y) \\ P + Q &\stackrel{\text{def}}{=} \nu a\,((\star a(x).x[in_1].x[].0 \mid \star a(x).x[in_2].x[].0) \mid \star a[x].\mathsf{case}\ x.\{x().P; x().Q\}) \end{aligned}$$

*Semantics.* In the presence of access points, it makes sense to add a garbage collection rule to the structural congruence to dispense with unused cut variables.

$$\nu x\vec{x}\,(P \mid Q) \equiv \nu\vec{x}\,(P \mid Q), \quad \text{if } x \notin fv(P) \cup fv(Q)$$

The reduction for ! against ? need not operate across the cut at which the access point is bound, and the channel is not discarded as the access point can be used an arbitrary number of times.

$$\nu\vec{x}\,(\star a(x).P \mid \star a[x].Q) \longrightarrow_{\mathrm{C}} \nu\vec{x}x\,(P \mid Q)$$

In order to give a closed system of reduction rules we assume MultiCut. We disable the structural congruence rules for explicitly performing replication and garbage collection of ! channels, as these are no longer necessary due to weakening and contraction on !.

The commuting conversions for access points are unsurprising.

$$\begin{aligned} \nu\vec{z}\,(\star x(y).P \mid Q) &\longrightarrow_{\mathrm{CC}} \star x(y).\nu\vec{z}\,(P \mid Q) \\ \nu\vec{z}\,(\star x[y].P \mid Q) &\longrightarrow_{\mathrm{CC}} \star x[y].\nu\vec{z}\,(P \mid Q) \end{aligned}$$

CP extended with access points does not enjoy termination, deadlock-freedom, or determinism. In return for losing these properties, access points significantly

increase the expressiveness of CP. Indeed, the stateful nondeterminism inherent in the semantics of access points also admits a meaningful notion of racy communication. A critical feature that CP lacks, even if we add multicut and nondeterminism, is any form of concurrent shared state. For instance, it is not possible to represent the standard session typing example of a book seller in CP in such a way that different buyers can observe any information about which books have been sold by the book seller to other buyers. Plain replication only allows us to copy the entire book seller. Access points do allow us to implement such examples.

*Example: State.* We implement a state cell of type $A$ as an access point of type $\star A$.

$$\begin{aligned} \mathit{State}\ A &\stackrel{\text{def}}{=} \star A \\ \mathit{new}(v).(\nu r)P &\stackrel{\text{def}}{=} (\nu r)(\star r(x).x \leftrightarrow v \mid P) \\ \mathit{put}(r, v).P &\stackrel{\text{def}}{=} \star r[v'].(\star r(x).x \leftrightarrow v \mid P) \\ \mathit{get}(r).(\nu v)P &\stackrel{\text{def}}{=} \star r[v].(\star r(x).x \leftrightarrow v \mid P) \end{aligned}$$

The *new* operation allocates a new reference cell ($r$) initialised to the supplied value. The *put* operation discards the existing state ($v'$). The *get* operation duplicates the existing state ($v$).

Using this encoding of state we can construct Landin's knot [Landin, 1964], which in sequential languages shows how higher-order state entails recursion. In CP, this provides a way of implementing recursive and replicated processes. For example, a nonterminating process can be defined by

$$\mathit{forever} \stackrel{\text{def}}{=} \nu f\ (\mathit{id}(f) \mid \mathit{new}(f).(\nu r)\nu g\ (\mathit{suspend}(g, \mathit{GF}(r)) \mid \mathit{put}(r, g).\mathit{GF}(r)))$$

where

$$\begin{aligned} \mathit{suspend}(f, P) &\stackrel{\text{def}}{=} \star f(x).P \\ \mathit{force}(f) &\stackrel{\text{def}}{=} \star f[x].0 \\ \mathit{id}(f) &\stackrel{\text{def}}{=} \mathit{suspend}(f, 0) \\ \mathit{GF}(r) &\stackrel{\text{def}}{=} \mathit{get}(r).(\nu f)\mathit{force}(f) \end{aligned}$$

The contents of the reference cell has type $\star(\star A)$, where $A$ can by any type.

More usefully, we can implement replication in the same way as *forever*, except the replicated process is placed in parallel with the recursive call represented by the body of the suspension.

$$\begin{aligned} !x(y).P \stackrel{\text{def}}{=} \nu f\ (\mathit{id}(f) \mid \mathit{new}(f).(\nu r)\nu g\ (&\ \mathit{suspend}(g, (\mathit{GF}(r) \mid \star x(y).P)) \\ &\mid \mathit{put}(r, g).\mathit{GF}(r))) \end{aligned}$$

*Example: Phil's bookstore.* As a simple example of using access points to capture concurrent state, we model a scenario in which Phil has three books: two copies of *Introduction to Functional Programming* [Bird and Wadler, 1988] and one

copy of *Java Generics and Collections* [Naftalin and Wadler, 2006], which he is willing to give to his students.

$$\begin{aligned} \mathit{Phil} &\stackrel{\text{def}}{=} !\mathit{phil}(x).\mathsf{case}\ x.\{\mathit{FP} \to \star\mathit{fp}[y].x \leftrightarrow y; \mathit{JG} \to \star\mathit{jg}[y].x \leftrightarrow y\} \\ \mathit{Fp} &\stackrel{\text{def}}{=} \star\mathit{fp}(x).x[\mathit{Yes}].\star\mathit{fp}(x).x[\mathit{Yes}].\star\mathit{fp}(x).x[\mathit{No}].0 \\ \mathit{Jg} &\stackrel{\text{def}}{=} \star\mathit{jg}(x).x[\mathit{Yes}].\star\mathit{jg}(x).x[\mathit{No}].0 \end{aligned}$$

For clarity, we tag the sums, as we did for booleans in (§3). The *Phil* process accepts communications on the *phil* channel, dispatching requests for *Java Generics and Collections* to the *Jg* process and requests for *Introduction to Functional Programming* to the *Fp* process. The types of the access point channels are as follows.

$$\begin{aligned} \mathit{phil} &: \star(\mathit{JG} : (\mathit{Yes} : \star 0 \oplus \mathit{No} : \star 0)\ \&\ \mathit{FP} : (\mathit{Yes} : \star 0 \oplus \mathit{No} : \star 0)\} \\ \mathit{fp} &: \star(\mathit{Yes} : \star 0 \oplus \mathit{No} : \star 0) \\ \mathit{jg} &: \star(\mathit{Yes} : \star 0 \oplus \mathit{No} : \star 0) \end{aligned}$$

We exploit the type $\star 0$ to allow processes to be implicitly terminated. Now let us define processes to represent Phil's PhD students.

$$\begin{aligned} \mathit{Jack} &\stackrel{\text{def}}{=} \star\mathit{phil}[x].x[\mathit{JG}].\mathsf{case}\ x.\{\mathit{Yes} \to \mathit{JackHappy}; \mathit{No} \to \mathit{JackSad}\} \\ \mathit{Jakub} &\stackrel{\text{def}}{=} \star\mathit{phil}[x].x[\mathit{JG}].\mathsf{case}\ x.\{\mathit{Yes} \to \mathit{JakubHappy}; \mathit{No} \to \mathit{JakubSad}\} \\ \mathit{Shayan} &\stackrel{\text{def}}{=} \star\mathit{phil}[x].x[\mathit{FP}].\mathsf{case}\ x.\{\mathit{Yes} \to \mathit{ShayanHappy}; \mathit{No} \to \mathit{ShayanSad}\} \\ \mathit{Simon} &\stackrel{\text{def}}{=} \star\mathit{phil}[x].x[\mathit{FP}].\mathsf{case}\ x.\{\mathit{Yes} \to \mathit{SimonHappy}; \mathit{No} \to \mathit{SimonSad}\} \end{aligned}$$

Jack and Jakub request *Java Generics and Collections*. Shayan and Simon request *Introduction to Functional Programming*. We can compose all of these processes in parallel.

$$(\mathit{Phil} \mid \mathit{Jg} \mid \mathit{Fp}) \mid \mathit{Jack} \mid \mathit{Jakub} \mid \mathit{Shayan} \mid \mathit{Simon}$$

Reduction will nondeterministically assign each book to a student, until at some point Phil runs out of the *Java Generics and Collections* book and either Jack or Jakub has a request denied, having to go to the library instead. Note that the choice of whether Jack or Jakub is sad is not made locally by any of the *Jack*, *Jakub* or *Phil* processes, but as a result of the racy interaction between them. Without access points it is not possible in CP to represent this kind of stateful and racy interaction.

*Do we need access points?* In classical linear logic, and hence CP, weakening and contraction are derivable for each type built up from the "negative" connectives ($\parr$, $\bot$, $\&$, $\top$, and ?). Once we conflate dual connectives, weakening and contraction also become derivable for each type built up from corresponding the positive connectives as well.

Given that conflating duals allows us to define weakening and contraction at all types, even in the multiplicative additive fragment of CP (without ! and

?), one might suspect that this would result in the effective identification of ! and ?, and hence we should be able to define unlimited state without using access points. This suggests a contradiction, because we claim that the resulting calculus is terminating.

In fact, we can simulate a form of unlimited state in this manner, but with a caveat. The implementation of weakening and contraction in this encoding is explicit. Correspondingly, the size of the definition of each operation is at least linear in the number of times the reference cell is accessed. Non-terminating programs written using Landin's knot necessarily access the same reference cell an infinite number of times. Hence, a process (without access points) encoding such a program would necessarily be infinite. Thus, we cannot encode full higher-order state and Landin's knot without access points.

## 5 Conclusions and Future Work

We have explored extensions to Wadler's CP calculus that make it more expressive, inspired syntactically by session types and the $\pi$-calculus and semantically by Abramsky et al.'s canonical interaction category **SCons**. In doing so, we have discovered an unexpected coincidence: that adding $\pi$-calculus terms "missing" from CP allows us to realise the identification of dual types present in **SCons**, while introducing bi-implications between the dual types allows us to derive the logical rules corresponding to the missing terms. Our approach seems to cover much of the expressivity gap between CP and $\pi$-calculus, including nondeterminism, concurrent state, and recursion. We conclude by sketching several future directions suggested by this work.

**SCons** has more structure than that necessary to express linear logic; in particular, individual **SCons** processes may have internal notions of state, and thus changing behaviour, without changing types. In contrast, CP (and other linear logic-inspired process calculi) capture all state explicitly in the types, and each copy of a replicated process always behave identically. This poses two questions. First, can an approach similar to interaction categories be adapted to describe channel passing behaviour? Second, how do notions of state, as encoded using access points (i.e., the identification of the dual access points) correspond to the notion of state internal to an **SCons** process?

Access points are rather a blunt tool. While they do yield concurrent state, they also destroy many of the nice meta-theoretic properties of CP. We believe that it should be possible to distil better behaved abstractions that still provide some of the extra expressiveness of access points.

While we believe that the extended calculus is sufficient to encode the (typed) $\pi$-calculus, we think it would be valuable to demonstrate such a semantics-preserving encoding explicitly. We would also like to address quantification, both first-order (corresponding to value-passing calculi) and second-order (corresponding to polymorphic channel-passing calculi).

In recent work [Lindley and Morris, 2015], we demonstrate a tight correspondence between cut-reduction in CP and a small-step operational semantics

for GV, a linear lambda-calculus extended with primitives for session-typing. It would be interesting to extend that correspondence to include additional features such as access points. Moreover, we are also studying well-founded recursive and corecursive session types in CP and their relationship to inductive linear data types in GV. We conjecture that conflating well-founded recursive and corecursive session types in CP will yield non-well-founded recursive session types and hence a way of encoding the entirety of the untyped $\pi$-calculus through a universal session type.

*Acknowledgements.*
This work was supported by the EPSRC grant: *From Data Types to Session Types—A Basis for Concurrency and Distribution* (EP/K034413/1).

## References

Abramsky, S.: Proofs as processes. Theor. Comput. Sci. 135(1), 5–9 (Apr 1992)
Abramsky, S., Gay, S.J., Nagarajan, R.: Interaction categories and the foundations of typed concurrent programming. In: Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany. pp. 35–113 (1996)
Bellin, G., Scott, P.J.: On the $\pi$-Calculus and linear logic. Theoretical Computer Science 135(1), 11–65 (1994)
Bird, R., Wadler, P.: An Introduction to Functional Programming. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK (1988)
Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: CONCUR. Springer (2010)
Curry, H.B.: Functionality in combinatory logic. Proceedings of the National Academy of Science 20, 584–590 (1934)
Dardha, O., Pérez, J.A.: Comparing deadlock-free session typed processes. In: EXPRESS/SOS, 2015, Madrid, Spain, 31st August 2015. pp. 1–15. Madrid, Spain (2015)
Ehrhard, T., Laurent, O.: Interpreting a finitary pi-calculus in differential interaction nets. Inf. Comput. 208(6), 606–633 (2010), http://dx.doi.org/10.1016/j.ic.2009.06.005
Fiore, M.P.: Differential structure in models of multiplicative biadditive intuitionistic linear logic. In: Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings. pp. 163–177 (2007)
Giachino, E., Kobayashi, N., Laneve, C.: Deadlock analysis of unbounded process networks. In: CONCUR. Springer (2014)
Girard, J.Y.: Linear logic. Theoretical Computer Science 50(1), 1–101 (Jan 1987)
Honda, K.: Types for dyadic interaction. In: CONCUR. Springer (1993)
Honda, K., Laurent, O.: An exact correspondence between a typed pi-calculus and polarised proof-nets. Theor. Comput. Sci. 411(22-24), 2223–2238 (2010), http://dx.doi.org/10.1016/j.tcs.2010.01.028

Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: ESOP. Springer (1998)
Honda, K., Yoshida, N., Berger, M.: Control in the $\pi$-calculus. In: Fourth ACM-SIGPLAN Continuation Workshop, CW04, 2004. Online proceedings. (2004)
Houston, R.: Finite products are biproducts in a compact closed category. Journal of Pure and Applied Algebra 212(2), 394 – 400 (2008), http://www.sciencedirect.com/science/article/pii/S0022404907001454
Howard, W.A.: The formulae-as-types notion of construction. In: Seldin, J.P., Hindley, J.R. (eds.) To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism. Academic Press, Boston, MA (1980)
Kobayashi, N.: A new type system for deadlock-free processes. In: CONCUR. Springer (2006), http://dx.doi.org/10.1007/11817949_16
Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the $\pi$-calculus. In: POPL. ACM (1996)
Landin, P.J.: The mechanical evaluation of expressions. Computer Journal 6(4), 308–320 (1964)
Lindley, S., Morris, J.G.: A semantics for propositions as sessions. In: Vitek, J. (ed.) ESOP 2015. Lecture Notes in Computer Science, vol. 9032, pp. 560–584. Springer (2015), http://dx.doi.org/10.1007/978-3-662-46669-8_23
Mazza, D.: The true concurrency of differential interaction nets. Mathematical Structures in Computer Science (2015), to appear
Naftalin, M., Wadler, P.: Java Generics and Collections. O'Reilly Media, Inc. (2006)
Padovani, L.: Deadlock and lock freedom in the linear $\pi$-calculus. In: LICS. ACM (2014), http://doi.acm.org/10.1145/2603088.2603116
Tait, W.W.: Infinitely long terms of transfinite type. In: Crossley, J.N., Dummett, M.A.E. (eds.) Formal Systems and Recursive Functions. North-Holland, Amsterdam (1965)
Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: PARLE. Springer (1994)
Wadler, P.: Propositions as sessions. J. Funct. Program. 24(2-3), 384–418 (2014)
Wadler, P.: Propositions as types. Commun. ACM 58(12), 75–84 (2015), http://doi.acm.org/10.1145/2699407