Propositions as Sessions

Philip Wadler
University of Edinburgh
wadler@inf.ed.ac.uk

Abstract

Continuing a line of work by Abramsky (1994), by Bellin and Scott (1994), and by Caires and Pfenning (2010), among others, this paper presents CP, a calculus in which propositions of classical linear logic correspond to session types. Continuing a line of work by Honda (1993), by Honda, Kubo, and Vasconcelos (1998), and by Gay and Vasconcelos (2010), among others, this paper presents GV, a linear functional language with session types, and presents a translation from GV into CP. The translation formalises for the first time a connection between a standard presentation of session types and linear logic, and shows how a modification to the standard presentation yield a language free from deadlock, where deadlock freedom follows from the correspondence to linear logic.

Categories and Subject Descriptors F.4.1 [*Mathematical Logic*]: Lambda calculus and related systems; F.4.1 [*Mathematical Logic*]: Proof theory; D.3.3 [*Language Constructs and Features*]: Concurrent programming structures

Keywords linear logic, lambda calculus, pi calculus

1. Introduction

Functional programmers know where they stand: upon the foundation of λ-calculus. Its canonicality is confirmed by its double discovery, once as natural deduction by Gentzen and once as λ-calculus by Church. These two formulations are related by the Curry-Howard correspondence, which takes

propositions *as* types,
proofs *as* programs, and
normalisation of proofs *as* evaluation of programs.

The correspondence arises repeatedly: Girard's System F corresponds to Reynold's polymorphic λ-calculus; Peirce's law in classical logic corresponds to Felleisen's call-cc.

Today, mobile phones, server farms, and multicores make us all concurrent programmers. Where lies a foundation for concurrency as firm as that of λ-calculus? Many process calculi have emerged—ranging from CSP to CCS to π-calculus to join calculus to mobile ambients to bigraphs—but none is as canonical as λ-calculus, and none has the distinction of arising from Curry-Howard.

Since its inception by Girard (1987), linear logic has held the promise of a foundation for concurrency rooted in Curry-Howard. In an early step, Abramsky (1994) and Bellin and Scott (1994) devised a translation from linear logic into π-calculus. Along another line, Honda (1993) introduced session types, further developed by Honda et al. (1998) and others, which take inspiration from linear logic, but do not enjoy a relationship as tight as Curry-Howard.

Recently, Caires and Pfenning (2010) found a twist on Abramsky's translation that yields an interpretation strongly reminiscent of session types, and a variant of Curry-Howard with

propositions *as* session types,
proofs *as* processes, and
cut elimination *as* communication.

The correspondence is developed in a series of papers by Caires, Pfenning, Toninho, and Pérez. This paper extends these lines of work with three contributions.

First, inspired by the calculus πDILL of Caires and Pfenning (2010), this paper presents the calculus CP. Based on dual intuitionistic linear logic, πDILL uses two-sided sequents, with two constructs corresponding to output ($\otimes$ on the right of a sequent and $\multimap$ on the left), and two constructs corresponding to input ($\multimap$ on the right of a sequent and $\otimes$ on the left). Based on classical linear logic, CP uses one-sided sequents, offering greater simplicity and symmetry, with a single construct for output ($\otimes$) and a single construct for input ($\parr$), each dual to the other. Caires et al. (2012a) compares πDILL with πCLL, which like CP is based on classical linear logic; we discuss this comparison in Section 4. (If you like, CP stands for Classical Processes.)

Second, though πDILL is clearly reminiscent of the body of work on session types, no one has previously published a formal connection. Inspired by the linear functional language with session types of Gay and Vasconcelos (2010), this paper presents the calculus GV, and presents a translation from GV into CP, for the first time formalising a tight connection between a standard presentation of session types and linear logic. In order to facilitate the translation, GV differs from the language of Gay and Vasconcelos (2010) in some particulars. These differences suffice to make GV, unlike the original, free from deadlock. (If you like, GV stands for Good Variation.)

Curry-Howard relates proof normalisation to computation. Logicians devised proof normalisation to show consistency of logic, and for this purpose it is essential that proof normalisation terminates. Hence, a consequence of Curry-Howard is that it identifies a fragment of λ-calculus for which the Halting Problem is solved. Well-typed programs terminate unless they explicitly resort to non-logical features such as general recursion. Similarly, a consequence of Curry-Howard for concurrency is that it identifies a fragment of a process calculus which is free of deadlock. In particular, πDILL and CP are both such fragments, and the proof that GV is deadlock-free follows immediately from its translation to CP.

Third, this paper presents a calculus with a stronger connection to linear logic, at the cost of a weaker connection to traditional process calculi. Bellin and Scott (1994) and Caires and Pfenning (2010) each present a translation from linear logic into π-calculus such that cut elimination converts one proof to another if and only if the translation of the one reduces to the translation of the other; but to achieve this tight connection several adjustments are necessary.

Bellin and Scott (1994) restrict the axiom to atomic type, and Caires and Pfenning (2010) omit the axiom entirely. In terms of a practical programming language, such restrictions are excessive. The former permits type variables, but instantiating a type variable to a type requires restructuring the program (as opposed to simple substitution); the latter outlaws type variables altogether. In consequence, neither system lends itself to parametric polymorphism.

Further, Bellin and Scott (1994) only obtain a tight correspondence between cut elimination and π-calculus for the multiplicative connectives, and they require a variant of π-calculus with surprising structural equivalences such as $x(y).x(z).P \equiv x(z).x(y).P$—permuting two reads on the same channel! Caires and Pfenning (2010) only obtain a tight correspondence between cut elimination and π-calculus by ignoring commuting conversions; this is hard to justify logically, because commuting conversions play an essential role in cut elimination. Pérez et al. (2012) show commuting conversions correspond to contextual equivalences, but fail to capture the directionality of the commuting conversions.

Thus, while the connection established in previous work between cut elimination in linear logic and reduction in π-calculus is encouraging, it comes at a substantial cost. Accordingly, this paper cuts the Gordian knot: it takes the traditional rules of cut elimination as specifying the reduction rules for its process calculus. Pro: we let logic guide the design of the 'right' process calculus. Con: we forego the assurance that comes from double discoveries of the same system, as with Gentzen and Church, Girard and Reynolds, and Pierce and Felleisen. Mitigating the con slightly are the results cited above that show a connection between Girard's linear logic and Milner's π-calculus, albeit not as tight as the other connections just mentioned.

In return for loosening its connection to π-calculus, the design of CP avoids the problems described above. The axiom is interpreted at all types, using a construct suggested by Caires et al. (2012a), and consequently it is easy to extend the system to support polymorphism, using a construct suggested by Turner (1995). All commuting conversions of cut elimination are satisfied.

This paper is organised as follows. Section 2 presents CP. Section 3 presents GV and its translation to CP. Section 4 discusses related work. Section 5 concludes.

2. Classical linear logic as a process calculus

This section presents CP, a session-typed process calculus. CP is based on classical linear logic with one-sided sequents, the system treated in the first paper on linear logic by Girard (1987).

Types Propositions, which may be interpreted as session types, are defined by the following grammar:

$A, B, C ::=$	
X	propositional variable
$X^\perp$	dual of propositional variable
$A \otimes B$	'times', output A then behave as B
$A \parr B$	'par', input A then behave as B
$A \oplus B$	'plus', select from A or B
$A \with B$	'with', offer choice of A or B
$!A$	'of course!', server accept
$?A$	'why not?', client request
$\exists X.B$	existential, output a type
$\forall X.B$	universal, input a type
1	unit for $\otimes$
$\perp$	unit for $\parr$
0	unit for $\oplus$
$\top$	unit for $\with$

Let A, B, C range over propositions, and X, Y, Z range over propositional variables. Every propositional variable X has a dual written $X^\perp$. Propositions are composed from multiplicatives ($\otimes$, $\parr$), additives ($\oplus$,$\with$), exponentials ($!$, $?$), second-order quantifiers ($\exists, \forall$), and units ($1, \perp, 0, \top$). In $\exists X.B$ and $\forall X.B$, propositional variable X is bound in B. Write $\mathsf{fv}(A)$ for the free variables in proposition A.

Duals Duals play a key role, ensuring that a request for input at one end of a channel matches an offer of a corresponding output at the other, and that a request to make a selection at one end matches an offer of a corresponding choice at the other.

Each proposition A has a dual $A^\perp$, defined as follows:

$$\begin{array}{rclcrcl}
(X)^\perp &=& X^\perp & & (X^\perp)^\perp &=& X \\
(A \otimes B)^\perp &=& A^\perp \parr B^\perp & & (A \parr B)^\perp &=& A^\perp \otimes B^\perp \\
(A \oplus B)^\perp &=& A^\perp \with B^\perp & & (A \with B)^\perp &=& A^\perp \oplus B^\perp \\
(!A)^\perp &=& ?A^\perp & & (?A)^\perp &=& !A^\perp \\
(\exists X.B)^\perp &=& \forall X.B^\perp & & (\forall X.B)^\perp &=& \exists X.B^\perp \\
1^\perp &=& \perp & & \perp^\perp &=& 1 \\
0^\perp &=& \top & & \top^\perp &=& 0
\end{array}$$

The dual of a propositional variable, $X^\perp$, is part of the syntax. Multiplicatives are dual to each other, as are additives, exponentials, and quantifiers.

Duality is an involution, $(A^\perp)^\perp = A$.

Substitution Write $B\{A/X\}$ to denote substitution of A for X in B. Substitution of a proposition for a dual propositional variable results in the dual of the proposition. Assuming $X \neq Y$, define

$$\begin{array}{rclcrcl}
X\{A/X\} &=& A & & X^\perp\{A/X\} &=& A^\perp \\
Y\{A/X\} &=& Y & & Y^\perp\{A/X\} &=& Y^\perp
\end{array}$$

The remaining clauses are entirely standard, for instance $(A \otimes B)\{C/X\} = A\{C/X\} \otimes B\{C/X\}$.

Duality preserves substitution, $B\{A/X\}^\perp = B^\perp\{A/X\}$.

Environments Let Γ, Δ, Θ range over environments associating names to propositions, where each name is distinct. Assume $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, with $x_i \neq x_j$ whenever $i \neq j$. Write $\mathsf{fn}(\Gamma) = \{x_1, \ldots, x_n\}$ for the names in Γ, and $\mathsf{fv}(\Gamma) = \mathsf{fv}(A_1) \cup \cdots \cup \mathsf{fv}(A_n)$ for the free variables in Γ. Order in environments is ignored. Environments use linear maintenance. Two environments may be combined only if they contain distinct names: writing Γ, Δ implies $\mathsf{fn}(\Gamma) \cap \mathsf{fn}(\Delta) = \emptyset$.

Processes Our process calculus is a variant on the π-calculus (Milner et al., 1992). Processes are defined by the following grammar:

$P, Q, R ::=$	
$x \leftrightarrow y$	link
$\nu x : A.(P \mid Q)$	parallel composition
$x[y].(P \mid Q)$	output
$x(y).P$	input
$x[\mathsf{inl}].P$	left selection
$x[\mathsf{inr}].P$	right selection
$x.\mathsf{case}(P, Q)$	choice
$!x(y).P$	server accept
$?x[y].P$	client request
$x[A].P$	send type
$x(X).P$	receive type
$x[\,].0$	empty output
$x().P$	empty input
$x.\mathsf{case}()$	empty choice

In $\nu x : A.(P \mid Q)$, name x is bound in P and Q, in $x[y].(P \mid Q)$, name y is bound in P (but not in Q), and in $x(y).P$, $?x[y].P$, and

$$\dfrac{}{w\leftrightarrow x \vdash w : A^{\perp},\, x : A}\ \mathsf{Ax} \qquad \dfrac{P \vdash \Gamma,\, x : A \quad Q \vdash \Delta,\, x : A^{\perp}}{\nu x : A.(P \mid Q) \vdash \Gamma,\, \Delta}\ \mathsf{Cut}$$

$$\dfrac{P \vdash \Gamma,\, y : A \quad Q \vdash \Delta,\, x : B}{x[y].(P \mid Q) \vdash \Gamma,\, \Delta,\, x : A \otimes B}\ \otimes \qquad \dfrac{R \vdash \Theta,\, y : A,\, x : B}{x(y).R \vdash \Theta,\, x : A \parr B}\ \parr$$

$$\dfrac{P \vdash \Gamma,\, x : A}{x[\mathsf{inl}].P \vdash \Gamma,\, x : A \oplus B}\ \oplus_1 \qquad \dfrac{P \vdash \Gamma,\, x : B}{x[\mathsf{inr}].P \vdash \Gamma,\, x : A \oplus B}\ \oplus_2 \qquad \dfrac{Q \vdash \Delta,\, x : A \quad R \vdash \Delta,\, x : B}{x.\mathsf{case}(Q, R) \vdash \Delta,\, x : A \,\&\, B}\ \&$$

$$\dfrac{P \vdash ?\Gamma,\, y : A}{!x(y).P \vdash ?\Gamma,\, x : !A}\ ! \qquad \dfrac{Q \vdash \Delta,\, y : A}{?x[y].Q \vdash \Delta,\, x : ?A}\ ? \qquad \dfrac{Q \vdash \Delta}{Q \vdash \Delta,\, x : ?A}\ \mathsf{Weaken} \qquad \dfrac{Q \vdash \Delta,\, x' : ?A,\, x'' : ?A}{Q\{x/x', x/x''\} \vdash \Delta,\, x : ?A}\ \mathsf{Contract}$$

$$\dfrac{P \vdash \Gamma,\, x : B\{A/X\}}{x[A].P \vdash \Gamma,\, x : \exists X.B}\ \exists \qquad \dfrac{Q \vdash \Delta,\, x : B}{x(X).Q \vdash \Delta,\, x : \forall X.B}\ \forall\ (X \notin \mathsf{fv}(\Delta))$$

$$\dfrac{}{x[\,].0 \vdash x : 1}\ 1 \qquad \dfrac{P \vdash \Gamma}{x().P \vdash \Gamma,\, x : \bot}\ \bot \qquad \text{(no rule for 0)} \qquad \dfrac{}{x.\mathsf{case}() \vdash \Gamma,\, x : \top}\ \top$$

Figure 1. CP, classical linear logic as a session-typed process calculus

$!x(y).P$, name y is bound in P. We write $\mathsf{fn}(P)$ for the free names in process P. In $x(X).P$, propositional variable X is bound in P.

The form $x\leftrightarrow y$ denotes forwarding, where every message received on x is retransmitted on y, and every message received on y is retransmitted on x. Square brackets surround outputs and round brackets surround inputs; unlike π-calculus, both output and input names are bound. The forms $x(y).P$ and $!x(y).P$ in our calculus behave like the same forms in π-calculus, while the forms $x[y].P$ and $?x[y].P$ in our calculus both behave like the form $\nu y.x\langle y\rangle.P$ in π-calculus.

A referee suggested, in line with one tradition for π-calculus, choosing the notation $\overline{x}(y).P$ in place of $x[y].P$; but overlines can be hard to spot, while the distinction between round and square brackets is clear.

Judgments The rules for assigning session types to processes are shown in Figure 1. Judgments take the form

$$P \vdash x_1 : A_1,\, \ldots,\, x_n : A_n$$

indicating that process P communicates along each channel named x_i obeying the protocol specified by A_i. Erasing the process and the channel names from the above yields

$$\vdash A_1,\, \ldots,\, A_n$$

and applying this erasure to the rules in Figure 1 yields the rules of classical linear logic, as given by Girard (1987).

2.1 Structural rules

The calculus has two structural rules, Axiom and Cut. We do not list Exchange explicitly, since order in environments is ignored.

The axiom is:

$$\dfrac{}{w\leftrightarrow x \vdash w : A^{\perp},\, x : A}\ \mathsf{Ax}$$

We interpret the axiom as forwarding. A name input along w is forwarded as output along x, and vice versa, so types of the two channels must be dual. Bellin and Scott (1994) restrict the axiom to propositional variables, replacing A by X and replacing $w\leftrightarrow x$ by the π-calculus term $w(y).x\langle y\rangle.0$. Whereas we forward any number of times and in either direction, they forward only once and from X to $X^{\perp}$.

The cut rule is:

$$\dfrac{P \vdash \Gamma,\, x : A \quad Q \vdash \Delta,\, x : A^{\perp}}{\nu x : A.(P \mid Q) \vdash \Gamma,\, \Delta}\ \mathsf{Cut}$$

Following Abramsky (1994) and Bellin and Scott (1994), we interpret Cut as a symmetric operation combining parallel composition with name restriction. Process P communicates along channel x obeying protocol A, while process Q communicates along the same channel x obeying the dual protocol $A^{\perp}$. Duality guarantees that sends and selections in P match with receives and choices in Q, and vice versa. Communications along Γ and Δ are disjoint, so P and Q are restricted to communicate with each other only along x. If communication could occur along two channels rather than one, then one could form a loop of communications between P and Q that leads to deadlock.

Observe that, despite writing $\nu x : A$ in the syntax, the type of x differs in P and Q—it is A in the former but $A^{\perp}$ in the latter. Including the type A in the syntax for Cut guarantees that given the type of each free name in the term, each term has a unique type derivation. To save ink and eyestrain, the type is omitted when it is clear from the context.

Cut elimination corresponds to process reduction. Figure 2 shows three equivalences on cuts, and one reduction that simplifies a cut against an axiom, each specified in terms of derivation trees; from which we read off directly the corresponding equivalence or reduction on processes. We write $\equiv$ for equivalences and $\Longrightarrow$ for reductions. Equivalence (Swap) states that a cut is symmetric:

$$\nu x : A.(P \mid Q) \equiv \nu x : A^{\perp}.(Q \mid P)$$

It serves the same role as the π-calculus structural equivalenc for symmetry, $P \mid Q \equiv Q \mid P$. Equivalences (Assoc) permit reordering cuts:

$$\nu y.(\nu x.(P \mid Q) \mid R) \equiv \nu x.(P \mid \nu y.(Q \mid R))$$

It serves the same role as the π-calculus structural equivalences for associativity, $P \mid Q \equiv Q \mid P$, and scope extrusion $(\nu x.P) \mid Q \equiv \nu x.(P \mid Q)$ when $x \notin P$.

Reduction (AxCut) simplifies a cut against an axiom.

$$\nu x.(w\leftrightarrow x \mid P) \Longrightarrow P\{w/x\}$$

We write $P\{w/x\}$ to denote substitution of w for x in P.

$$\text{(Swap)}\qquad \dfrac{P \vdash \Gamma,\, x:A \quad Q \vdash \Delta,\, x:A^\perp}{\nu x:A.(P \mid Q) \vdash \Gamma,\, \Delta}\,\text{Cut} \quad\equiv\quad \dfrac{Q \vdash \Delta,\, x:A^\perp \quad P \vdash \Gamma,\, x:A}{\nu x:A^\perp.(Q \mid P) \vdash \Gamma,\, \Delta}\,\text{Cut}$$

$$\text{(Assoc)}\qquad \dfrac{\dfrac{P \vdash \Gamma,\, x:A \quad Q \vdash \Delta,\, x:A^\perp,\, y:B}{\nu x.(P \mid Q) \vdash \Gamma,\, \Delta,\, y:B}\,\text{Cut} \quad R \vdash \Theta,\, y:B^\perp}{\nu y.(\nu x.(P \mid Q) \mid R) \vdash \Gamma,\, \Delta,\, \Theta}\,\text{Cut} \quad\equiv\quad \dfrac{P \vdash \Gamma,\, x:A \quad \dfrac{Q \vdash \Delta,\, x:A^\perp,\, y:B \quad R \vdash \Theta,\, y:B^\perp}{\nu y.(Q \mid R) \vdash \Delta,\, \Theta,\, x:A^\perp}\,\text{Cut}}{\nu x.(P \mid \nu y.(Q \mid R)) \vdash \Gamma,\, \Delta,\, \Theta}\,\text{Cut}$$

$$\text{(AxCut)}\qquad \dfrac{\dfrac{}{w{\leftrightarrow}x \vdash w:A^\perp,\, x:A}\,\text{Ax} \quad P \vdash \Gamma,\, x:A^\perp}{\nu x.(w{\leftrightarrow}x \mid P) \vdash \Gamma,\, w:A^\perp}\,\text{Cut} \quad\Longrightarrow\quad P\{w/x\} \vdash \Gamma,\, w:A^\perp$$

Figure 2. Structural cut equivalences and reduction for CP

2.2 Output and input

The multiplicative connectives $A \otimes B$ and $A \parr B$ are dual. We interpret $A \otimes B$ as the session type of a process which outputs an A and then behaves as a B, and $A \parr B$ as the session type of a process which inputs an A and then behaves as a B.

The rule for output is:

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

Processes P and Q act on disjoint sets of channels. Process P communicates along channel y obeying protocol A, while process Q communicates along channel x obeying protocol B. The composite process $x[y].(P \mid Q)$ communicates along channel x obeying protocol $A \otimes B$; it allocates a fresh channel y, transmits y along x, and then concurrently executes P and Q. Disjointness of P and Q ensures there is no further entangling between x and y, which guarantees freedom from deadlock.

The rule for input is:

$$\dfrac{R \vdash \Theta,\, y:A,\, x:B}{x(y).R \vdash \Theta,\, x:A \parr B}\,\parr$$

Process R communicates along channel y obeying protocol A and along channel x obeying protocol B. The composite process $x(y).R$ communicates along channel x obeying protocol $A \parr B$; it receives name y along x, and then executes R. Unlike with output, the single process R that communicates with both x and y. It is safe to permit the same process to communicate with x and y on the input side, because there is no further entangling of x with y on the output side, explaining the claim that disentangling x from y on output guarantees freedom from deadlock.

For output, channel x has type B in the component process Q but type $A \otimes B$ in the composite process $x[y].(P \mid Q)$. For input, channel x has type B in the component process R but type $A \parr B$ in the composite process $x(y).R$. One may regard the type of the channel evolving as communication proceeds, corresponding to the notion of session type. Assigning the same channel name different types in the hypothesis and conclusion of a rule is the telling twist added by Caires and Pfenning (2010), in contrast to the use of different variables in the hypothesis and conclusion followed by Abramsky (1994) and Bellin and Scott (1994).

The computational content of the logic is most clearly revealed in the principal cuts of each connective against its dual. Principal cut reductions are shown in Figure 3.

Cut of output $\otimes$ against input $\parr$ corresponds to communication, as shown in rule $(\beta_{\otimes\parr})$:

$$\nu x.(x[y].(P \mid Q) \mid x(y).R) \Longrightarrow \nu y.(P \mid \nu x.(Q \mid R))$$

In stating this rule, we take advantage of the fact that y is bound in both $x[y].P$ and $x(y).Q$ to assume the same bound name y has been chosen in each; Pitts (2011) refers to this as the 'anti-Barendregt' convention.

Recall that $x[y].P$ in our notation corresponds to $\nu y.x\langle y\rangle.P$ in π-calculus. Thus, the rule above corresponds to the π-calculus reduction:

$$\nu x.(\nu y.x\langle y\rangle.(P \mid Q) \mid x(z).R) \Longrightarrow \nu y.P \mid \nu x.(Q \mid R\{z/y\})$$

This follows from from $x\langle y\rangle.P \mid x(z).R \Longrightarrow P \mid R\{z/y\}$, and the structural equivalences for scope extrusion, since $x \notin \mathsf{fn}(P)$.

The right-hand side of the above reduction can be written in two ways, which are equivalent by use of the the structural rules (Swap) and (Assoc).

$$\begin{array}{lll} & \nu x:A.(P \mid \nu y:B.(Q \mid R)) & \\ \equiv & \nu x:A.(P \mid \nu y:B^\perp.(R \mid Q)) & \text{(Swap)} \\ \equiv & \nu y:B^\perp.(\nu x:A.(P \mid R) \mid Q) & \text{(Assoc)} \\ \equiv & \nu y:B.(Q \mid \nu x:A.(P \mid R)) & \text{(Swap)} \end{array}$$

The apparent lack of symmetry between $A \otimes B$ and $B \otimes A$ may appear unsettling: the first means output A and then behave as B, the second means output B and then behave as A. The situation is similar to Cartesian product, where $B \times A$ and $A \times B$ differ but satisfy an isomorphism. Similarly, $A \otimes B$ and $B \otimes A$ are interconvertible.

$$\dfrac{\dfrac{\dfrac{}{w{\leftrightarrow}z \vdash w:B^\perp,\, z:B}\,\text{Ax} \quad \dfrac{}{y{\leftrightarrow}x \vdash y:A^\perp,\, x:A}\,\text{Ax}}{x[z].(w{\leftrightarrow}z \mid y{\leftrightarrow}x) \vdash w:B^\perp,\, y:A^\perp,\, x:B \otimes A}\,\otimes}{w(y).x[z].(w{\leftrightarrow}z \mid y{\leftrightarrow}x) \vdash w:A^\perp \parr B^\perp,\, x:B \otimes A}\,\parr$$

Let $Q \vdash \Delta$ be the conclusion of the above derivation. Given an arbitrary derivation ending in $P \vdash \Gamma,\, w:A \otimes B$, one may replace $A \otimes B$ with $B \otimes A$ as follows:

$$\dfrac{P \vdash \Gamma,\, w:A \otimes B \quad Q \vdash \Delta}{\nu w.(P \mid Q) \vdash \Gamma,\, x:B \otimes A}\,\text{Cut}$$

Here process P communicates along w obeying the protocol $A \otimes B$, outputting A and then behaving as B. Composing P with Q yields the process that communicates along x obeying the protocol $B \otimes A$, outputting B and then behaving as A.

The multiplicative units are 1 for $\otimes$ and $\perp$ for $\parr$. We interpret 1 as the session type of a process that performs an empty ouput, and $\perp$ as the session type of a process that performs an empty input. These are related by duality: $1^\perp = \perp$. Their rules are shown in Figure 1. Cut of empty output 1 against empty input $\perp$ corresponds to an empty communication, as shown in rule $(\beta_{1\perp})$:

$$\nu x.(x[\,].0 \mid x().P) \Longrightarrow P$$

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

$(\beta_{\oplus\with})$
$$\dfrac{\dfrac{P \vdash \Gamma, x : A}{x[\mathsf{inl}].P \vdash \Gamma, x : A \oplus B}\oplus_1 \quad \dfrac{Q \vdash \Delta, x : A^\perp \quad R \vdash \Delta, x : B^\perp}{x.\mathsf{case}(Q, R) \vdash \Delta, x : A^\perp \with B^\perp}\with}{\nu x.(x[\mathsf{inl}].P \mid x.\mathsf{case}(Q, R)) \vdash \Gamma, \Delta}\text{Cut} \Longrightarrow \dfrac{P \vdash \Gamma, x : A \quad Q \vdash \Delta, x : A^\perp}{\nu x.(P \mid Q) \vdash \Gamma, \Delta}\text{Cut}$$

$(\beta_{!?})$
$$\dfrac{\dfrac{P \vdash ?\Gamma, y : A}{!x(y).P \vdash ?\Gamma, x : !A}! \quad \dfrac{Q \vdash \Delta, y : A^\perp}{?x[y].Q \vdash \Delta, x : ?A^\perp}?}{\nu x.(!x(y).P \mid ?x[y].Q) \vdash ?\Gamma, \Delta}\text{Cut} \Longrightarrow \dfrac{P \vdash ?\Gamma, y : A \quad Q \vdash \Delta, y : A^\perp}{\nu y.(P \mid Q) \vdash ?\Gamma, \Delta}\text{Cut}$$

$(\beta_{!W})$
$$\dfrac{\dfrac{P \vdash ?\Gamma, y : A}{!x(y).P \vdash ?\Gamma, x : !A}! \quad \dfrac{Q \vdash \Delta}{Q \vdash \Delta, x : ?A^\perp}\text{Weaken}}{\nu x.(!x(y).P \mid Q) \vdash ?\Gamma, \Delta}\text{Cut} \Longrightarrow \dfrac{Q \vdash \Delta}{Q \vdash ?\Gamma, \Delta}\text{Weaken}$$

$(\beta_{!C})$
$$\dfrac{\dfrac{P \vdash ?\Gamma, y : A}{!x(y).P \vdash ?\Gamma, x : !A}! \quad \dfrac{Q \vdash \Delta, x' : ?A, x'' : ?A}{Q\{x/x', x/x''\} \vdash \Delta, x : ?A}\text{Contract}}{\nu x.(!x(y).P \mid Q\{x/x', x/x''\}) \vdash ?\Gamma, \Delta}\text{Cut} \Longrightarrow$$
$$\dfrac{\dfrac{\dfrac{P' \vdash ?\Gamma', y' : A}{!x'(y').P' \vdash ?\Gamma', x' : !A}! \quad \dfrac{\dfrac{P'' \vdash ?\Gamma'', y'' : A}{!x''(y'').P'' \vdash ?\Gamma'', x'' : !A}! \quad Q \vdash \Delta, x' : ?A^\perp, x'' : ?A^\perp}{\nu x''.(!x''(y'').P'' \mid Q) \vdash ?\Gamma', \Delta, x : ?A^\perp}\text{Cut}}{\nu x'.(!x'(y').P' \mid \nu x''.(!x''(y'').P'' \mid Q)) \vdash ?\Gamma', ?\Gamma'', \Delta}\text{Cut}}{\nu x'.(!x'(y).P \mid \nu x''.(!x''(y).P \mid Q)) \vdash ?\Gamma, \Delta}\text{Contract}$$

$(\beta_{\exists\forall})$
$$\dfrac{\dfrac{P \vdash \Gamma, x : B\{A/X\}}{x[A].P \vdash \Gamma, x : \exists X.B}\exists \quad \dfrac{Q \vdash \Delta, x : B^\perp}{x(X).Q \vdash \Delta, x : \forall X.B^\perp}\forall}{\nu x.(x[A].P \mid x(X).Q) \vdash \Gamma, \Delta}\text{Cut} \Longrightarrow \dfrac{P \vdash \Gamma, x : B\{A/X\} \quad Q\{A/X\} \vdash \Delta, x : B^\perp\{A/X\}}{\nu x.(P \mid Q\{A/X\}) \vdash \Gamma, \Delta}\text{Cut}$$

$(\beta_{1\perp})$
$$\dfrac{\dfrac{}{x[\,].0 \vdash x : 1}1 \quad \dfrac{P \vdash \Gamma}{x().P \vdash \Gamma, x : \perp}\perp}{\nu x.(x[\,].0 \mid x().P) \vdash \Gamma}\text{Cut} \Longrightarrow P \vdash \Gamma$$

$(\beta_{0\top})$ (no rule for 0 with $\top$)

Figure 3. Principal cut reductions for CP

This rule resembles reduction of a nilary communication in the polyadic π-calculus.

2.3 Selection and choice

The additive connectives $A \oplus B$ and $A \with B$ are dual. We interpret $A \oplus B$ as the session type of a process which selects from either an A or a B, and $A \with B$ as the session type of a process which offers a choice of either an A or a B.

The rule for left selection is:

$$\dfrac{P \vdash \Gamma, x : A}{x[\mathsf{inl}].P \vdash \Gamma, x : A \oplus B}\oplus_1$$

Process P communicates along channel x obeying protocol A. The composite process $x[\mathsf{inl}].P$ communicates along channel x obeying protocol $A \oplus B$; it transmits along x a request to select the left option from a choice, and then executes process P. The rule for right selection is symmetric.

The rule for choice is:

$$\dfrac{Q \vdash \Delta, x : A \quad R \vdash \Delta, x : B}{x.\mathsf{case}(Q, R) \vdash \Delta, x : A \with B}\with$$

The composite process $x.\mathsf{case}(Q, R)$ communicates along channel x obeying protocol $A \with B$; it receives a selection along channel x and executes either process Q or R accordingly.

For selection, channel x has type A in the component process P and type $A \oplus B$ in the composite process $x[\mathsf{inl}].P$. For choice, channel x has type A in the component process Q, type B in the component process R, and type $A \with B$ in the composite process $x.\mathsf{case}(Q, R)$. Again, one may regard the type of the channel evolving as communication proceeds, corresponding to the notion of session type.

$$
\begin{array}{llcll}
(\kappa_{\otimes 1}) & \nu z.(x[y].(P \mid Q) \mid R) & \Longrightarrow & x[y].(\nu z.(P \mid R) \mid Q), & \text{if } z \in \mathsf{fn}(P) \\
(\kappa_{\otimes 2}) & \nu z.(x[y].(P \mid Q) \mid R) & \Longrightarrow & x[y].(P \mid \nu z.(Q \mid R)), & \text{if } z \in \mathsf{fn}(Q) \\
(\kappa_{⅋}) & \nu z.(x(y).P \mid Q) & \Longrightarrow & x(y).\nu z.(P \mid Q) & \\
(\kappa_{\oplus}) & \nu z.(x[\mathsf{inl}].P \mid Q) & \Longrightarrow & x[\mathsf{inl}].\nu z.(P \mid Q) & \\
(\kappa_{\&}) & \nu z.(x.\mathsf{case}(P, Q) \mid R) & \Longrightarrow & x.\mathsf{case}(\nu z.(P \mid R), \nu z.(Q \mid R)) & \\
(\kappa_{!}) & \nu z.(!x(y).P \mid Q) & \Longrightarrow & !x(y).\nu z.(P \mid Q) & \\
(\kappa_{?}) & \nu z.(?x[y].P \mid Q) & \Longrightarrow & ?x[y].\nu z.(P \mid Q) & \\
(\kappa_{\exists}) & \nu z.(x[A].P \mid Q) & \Longrightarrow & x[A].\nu z.(P \mid Q) & \\
(\kappa_{\forall}) & \nu z.(x(X).P \mid Q) & \Longrightarrow & x(X).\nu z.(P \mid Q) & \\
(\kappa_{\perp}) & \nu z.(x().P \mid Q) & \Longrightarrow & x().\nu z.(P \mid Q) & \\
(\kappa_{0}) & & \text{(no rule for 0)} & & \\
(\kappa_{\top}) & \nu z.(x.\mathsf{case}() \mid Q) & \Longrightarrow & x.\mathsf{case}() &
\end{array}
$$

Figure 4. Commuting conversions for CP

Cut of selection $\oplus$ against choice $\&$ corresponds to picking an alternative, as shown in rule $(\beta_{\oplus\&})$:

$$x[\mathsf{inl}].P \mid x.\mathsf{case}(Q, R) \Longrightarrow \nu x.(P \mid Q)$$

The rule to select the right option is symmetric.

The additive units are 0 for $\oplus$ and $\top$ for $\&$. We interpret 0 as the session type of a process that selects from among no alternatives, and $\perp$ as the session type of a process that offers a choice among no alternatives. These are related by duality: $0^\perp = \top$. Their rules are shown in Figure 1. There is no rule for 0, because it is impossible to select from no alternatives. Hence, there is also no reduction for a cut of an empty selection against an empty choice, as shown in Figure 3.

2.4 Servers and clients

The exponential connectives $!A$ and $?A$ are dual. We interpret $!A$ as the session type of a server that will repeatedly accept an A, and interpret $?A$ as the session type of a collection of clients that may each request an A. Servers and clients are asymmetric: a server must be impartial, providing the same service to each client; whereas differing clients may accumulate requests to pass to the same server.

The rule for servers is:

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

Process P communicates along channel y obeying protocol A. The composite process $!x(y).P$ communicates along channel x obeying the protocol $!A$; it receives y along x, and then spawns a fresh copy of P to execute. All channels used by P other than y must obey a protocol of the form $?B$, for some B, to ensure that replicating P respects the type discipline. Intuitively, a process may only provide a replicable service if it is implemented by communicating only with other processes that provide replicable services.

There are three rules for clients, corresponding to the fact that a server may have one, none, or many clients. The three rules correspond to the standard rules of classical linear logic for dereliction, weakening, and contraction.

The first rule is for a single client.

$$\frac{Q \vdash \Delta,\, y : A}{?x[y].Q \vdash \Delta,\, x : ?A}\ ?$$

Process Q communicates along channel y obeying protocol A. The composite process $?x[y].Q$ communicates along channel x obeying protocol $?A$; it allocates a fresh channel y, transmits y along x, and then executes process Q. Cut of rule ! against rule ? corresponds to spawning a single copy of a server to communicate with a client, as shown in rule $(\beta_{!?})$:

$$\nu x.(!x(y).P \mid ?x[y].Q) \Longrightarrow \nu y.(P \mid Q)$$

The second rule is for no clients.

$$\frac{Q \vdash \Delta}{Q \vdash \Delta,\, x : ?A}\ \mathsf{Weaken}$$

A process Q that does not communicate along any channel obeying protocol A may be regarded as communicating along a channel obeying protocol $?A$. Cut of rule ! against Weaken corresponds to garbage collection, deallocating a server that has no clients, as shown in rule $(\beta_{!W})$:

$$\nu x.(!x(y).P \mid Q) \Longrightarrow Q, \quad \text{if } x \notin \mathsf{fn}(Q)$$

The third rule aggregates multiple clients.

$$\frac{Q \vdash \Delta,\, x' : ?A,\, x'' : ?A}{Q\{x/x', x/x''\} \vdash \Delta,\, x : ?A}\ \mathsf{Contract}$$

Process Q communicates along two channels x and x' both obeying protocol $?A$. Process $Q\{x/x', x/x''\}$ is identical to Q save all occurrences of x' and x'' have been renamed to x; it communicates along a single channel x obeying protocol $?A$. Cut of rule ! against Contract corresponds to replicating a server, as shown in rule $(\beta_{!C})$:

$$\begin{aligned}\nu x.(!x(y).P \mid Q\{x/x', x/x''\}) \Longrightarrow\\ \nu x'.(!x'(y).P \mid \nu x''.(!x''(y).P \mid Q))\end{aligned}$$

The type derivation on the right-hand side of rule $(\beta_{!C})$ applies Contract once for each free name z_i in Γ. The derivation is written using the following priming convention: we assume that to each name z there are associated other names z' and z'', and we write P' for the process identical to P save that each free name z in P has been replaced by z'; that is, if $\mathsf{fn}(P) = \{y, z_1, \ldots, z_n\}$ then $P' = P\{y'/y, z_1'/z_1, \ldots, z_n'/z_n\}$, and similarly for P''.

A referee notes weakening and contraction could be given explicit notation rather than implicit. For instance, we could replace Q by $?x[].Q$ in weakening, and $Q\{x/x', x/x''\}$ by $?x[x', x''].Q$ in contraction, yielding

$$\frac{Q \vdash \Delta}{?x[].Q \vdash \Delta,\, x : ?A}\ \mathsf{Weaken}$$

and

$$\frac{Q \vdash \Delta,\ x' : ?A,\ x'' : ?A}{?x[x', x''].Q \vdash \Delta,\ x : ?A}\ \textsf{Contract}$$

while reduction rules $(\beta_{!W})$ and $(\beta_{!C})$ become

$$\nu x.(!x(y).P \mid ?x[].Q) \Longrightarrow ?z_1[].\cdots.?z_n[].Q$$

and

$$\begin{aligned}&\nu x.(!x(y).P \mid ?x[x', x''].Q) \Longrightarrow\\ &\quad ?z_1[z_1', z_1''].\cdots.?z_n[z_n', z_n''].\\ &\qquad \nu x'.(!x'(y').P' \mid \nu x''.(!x''(y'').P'' \mid Q))\end{aligned}$$

where $\mathsf{fn}(P) = \{y, z_1, \ldots, z_n\}$.

2.5 Polymorphism

The quantifiers $\exists$ and $\forall$ are dual. We interpret $\exists X.B$ as the session type of a process that instantiates propositional variable X to a given proposition, and interpret $\forall X.B$ as the session type of a process that generalises over X. These correspond to type application and type abstraction in polymorphic λ-calculus, or to sending and receiving types in the polymorphic π-calculus of Turner (1995).

The rule for instantiation is:

$$\frac{P \vdash \Gamma,\ x : B\{A/X\}}{x[A].P \vdash \Gamma,\ x : \exists X.B}\ \exists$$

Process P communicates along channel x obeying protocol $B\{A/X\}$. The composite process $x[A].P$ communicates along channel x obeying protocol $\exists X.B$; it transmits a representation of A along x, and then executes P.

The rule for generalisation is:

$$\frac{Q \vdash \Delta,\ x : B}{x(X).Q \vdash \Delta,\ x : \forall X.B}\ \forall\ (X \notin \mathsf{fv}(\Delta))$$

Process Q communicates along channel x obeying protocol B. The composite process $x(X).Q$ communicates along channel x obeying protocol $\forall X.B$; it receives a description of a proposition along channel x, binds the proposition to the propositional variable X, and then executes Q.

Cut of instantiation $\exists$ against generalisation $\forall$ corresponds to transmitting a representation of a proposition, as shown in rule $(\beta_{\exists\forall})$:

$$\nu x.(x[A].P \mid x(X).Q) \Longrightarrow \nu x.(P \mid Q\{A/X\})$$

This rule behaves similarly to beta reduction of a type abstraction against a type application in polymorphic λ-calculus, or communication of a type in the polymorphic π-calculus.

2.6 Commuting conversions

Commuting conversions are shown in Figure 4. To save space, these are shown as reductions on terms, without the accompanying derivation trees.

Each commuting conversion pushes a cut inside a communication operation. There are two conversions for $\otimes$, depending upon whether the cut pushes into the left or right branch. Each of the remaining logical operators has one conversion, with the exception of $\oplus$, which has two (only the left rule is shown, the right rule is symmetric); and of 0, which has none.

An important aspect of CP is revealed by considering rule $(\kappa_{\invamp})$, which pushes cut inside input:

$$\nu z.(x(y).P \mid Q) \Longrightarrow x(y).\nu z.(P \mid Q)$$

On the left-hand side process Q may interact with the environment, while on the right-hand side Q is guarded by the input and cannot interact with the environment. In our setting, this is not problematic. If x is bound by an outer cut, then the guarding input is guaranteed to match a corresponding output at some point. If x is not bound by an outer cut, then we consider the process halted while it awaits external communication along x; compare this with the use of labeled transitions in Lemma 5.7 of Caires and Pfenning (2010).

2.7 Cut elimination

In addition to the rules of Figures 2, 3, and 4, we add a rule relating reductions to structural equivalences:

$$\frac{P \equiv Q \quad Q \Longrightarrow R \quad R \equiv S}{P \equiv S}$$

And we add rules that permit reduction under cut:

$$\frac{P \Longrightarrow R}{\nu x.(P \mid Q) \Longrightarrow \nu x.(R \mid Q)} \qquad \frac{Q \Longrightarrow R}{\nu x.(P \mid Q) \Longrightarrow \nu x.(R \mid Q)}$$

We do not add reduction under other operators; see below.

CP satisfies subject reduction: well-typed processes reduce to well-typed processes.

THEOREM 1. *If $P \vdash \Gamma$ and $P \Longrightarrow Q$ then $Q \vdash \Gamma$.*

Proof sketch: Figures 2 and 3 contain the relevant proofs for their rules, the proofs for Figure 4 are similar. □

Say process P is a *cut* if it has the form $\nu x.(Q \mid R)$ for some x, Q, and R. CP satisfies top-level cut elimination: every process reduces to a process that is not a cut.

THEOREM 2. *If $P \vdash \Gamma$ then there exists a Q such that $P \Longrightarrow^* Q$ and Q is not a cut.*

Proof sketch: If P is a cut, there are three possibilities. If one side of the cut uses the axiom, apply AxCut. If one side of the cut is itself a cut, recursively eliminate the cut. In the remaining cases, either both sides are logical rules that act on the cut variable, in which case a principal reduction of Figure 3 applies, or at least one side is a logical rule acting on a variable other than the cut variable, in which case a commuting reduction of Figure 4 applies. Since we support impredicative polymorphism, where a polymorphic type may be instantiated by a polymorphic type, some care is required in formulating the induction to ensure termination, but this is standard (Gallier, 1990). □

This result resembles the Principal Lemma of Cut Elimination (Girard et al., 1989, Section 13.2), which eliminates a final cut rule, possibly replacing it with (smaller) cuts further up the proof tree. Top-level cut elimination corresponds to lack of deadlock; it ensures that any process can reduce until it needs to perform an external communication.

If our goal was to eliminate all cuts, we would need to introduce congruence rules, such as

$$\frac{P \Longrightarrow Q}{x(y).P \Longrightarrow x(y).Q}$$

and similarly for each operator. Such rules do not correspond well to our notion of computation on processes, so we omit them; this is analogous to the usual practice of not permitting reduction under lambda.

3. A session-typed functional language

This section presents GV, a session-typed functional language based on one devised by Gay and Vasconcelos (2010), and presents its translation into CP.

Our presentation of GV differs in some particulars from that of Gay and Vasconcelos (2010). Most notably, our system is guaranteed free from deadlock whereas theirs is not. Achieving this property requires some modifications to their system. We split their session type 'end' into two dual types '$\mathsf{end}_!$' and '$\mathsf{end}_?$', and we

$$\frac{}{x : T \vdash x : T}\ \textsf{Id} \qquad \frac{}{\vdash \textsf{unit} : \textsf{Unit}}\ \textsf{Unit} \qquad \frac{\Phi \vdash N : U \quad \textsf{un}(T)}{\Phi,\ x : T \vdash N : U}\ \textsf{Weaken} \qquad \frac{\Phi,\ x' : T,\ x'' : T \vdash N : U \quad \textsf{un}(T)}{\Phi,\ x : T \vdash N\{x/x', x/x''\} : U}\ \textsf{Contract}$$

$$\frac{\Phi,\ x : T \vdash N : U}{\Phi \vdash \lambda x.\, N : T \multimap U}\ \multimap\textsf{-I} \qquad \frac{\Phi \vdash L : T \multimap U \quad \Psi \vdash M : T}{\Phi,\ \Psi \vdash L\ M : U}\ \multimap\textsf{-E} \qquad \frac{\Phi \vdash L : T \multimap U \quad \textsf{un}(\Phi)}{\Phi \vdash L : T \to U}\ \to\textsf{-I} \qquad \frac{\Phi \vdash L : T \to U}{\Phi \vdash L : T \multimap U}\ \to\textsf{-E}$$

$$\frac{\Phi \vdash M : T \quad \Psi \vdash N : U}{\Phi,\ \Psi \vdash (M, N) : T \otimes U}\ \otimes\textsf{-I} \qquad \frac{\Phi \vdash M : T \otimes U \quad \Psi,\ x : T,\ y : U \vdash N : V}{\Phi,\ \Psi \vdash \textsf{let}\ (x, y) = M\ \textsf{in}\ N : V}\ \otimes\textsf{-E}$$

$$\frac{\Phi \vdash M : T \quad \Psi \vdash N : !T.S}{\Phi,\ \Psi \vdash \textsf{send}\ M\ N : S}\ \textsf{Send} \qquad \frac{\Phi \vdash M : ?T.S}{\Phi \vdash \textsf{receive}\ M : T \otimes S}\ \textsf{Receive}$$

$$\frac{\Phi \vdash M : \oplus\{l_i : S_i\}_{i \in I}}{\Phi \vdash \textsf{select}\ l_j\ M : S_j}\ \textsf{Select} \qquad \frac{\Phi \vdash M : \&\{l_i : S_i\}_{i \in I} \quad (\Psi,\ x : S_i \vdash N_i : T)_{i \in I}}{\Phi,\ \Psi \vdash \textsf{case}\ M\ \textsf{of}\ \{l_i : x.N_i\}_{i \in I} : T}\ \textsf{Case}$$

$$\frac{\Phi,\ x : S \vdash M : \textsf{end}_! \quad \Psi,\ x : \overline{S} \vdash N : T}{\Phi,\ \Psi \vdash \textsf{with}\ x\ \textsf{connect}\ M\ \textsf{to}\ N : T}\ \textsf{Connect} \qquad \frac{\Phi \vdash M : T \otimes \textsf{end}_?}{\Phi \vdash \textsf{terminate}\ M : T}\ \textsf{Terminate}$$

Figure 5. GV, a session-typed functional language

replace their constructs ‘accept’, ‘request’, and ‘fork’, by two new constructs ‘with-connect-to’ and ‘terminate’.

A number of features of Gay and Vasconcelos (2010) are not echoed here. Their system is based on asynchronous buffered communication, they show that the size required of asynchronous buffers can be bounded by analysing session types, and they support recursive functions, recursive session types, and subtyping. We omit these contributions for simplicity, but see no immediate difficulty in extending our results to include them. Of course, adding recursive terms or recursive session types may remove the property that all programs terminate.

For simplicity, we also omit a number of other possible features. We do not consider base types, which are straightforward. We also do not consider how to add replicated servers with multiple clients, along the lines suggested by ! and ? in CP, or how to add polymorphism, along the lines suggested by $\exists$ and $\forall$ in CP, but both extensions appear straightforward.

Session types Session types are defined by the following grammar:

$S ::=$	
$!T.S$	output value of type T then behave as S
$?T.S$	input value of type T then behave as S
$\oplus\{l_i : S_i\}_{i \in I}$	select from behaviours S_i with label l_i
$\&\{l_i : S_i\}_{i \in I}$	offer choice of behaviours S_i with label l_i
$\textsf{end}_!$	terminator, convenient for use with output
$\textsf{end}_?$	terminator, convenient for use with input

Let S range over session types, and let T, U, V range over types. Session type $!T.S$ describes a channel along which a value of type T may be sent and which subsequently behaves as S. Dually, $?T.S$ describes a channel along which a value of type T may be received and which subsequently behaves as S. Session type $\oplus\{l_i : S_i\}_{1 \in I}$ describes a channel along which one of the distinct labels l_i may be sent and which subsequently behaves as S_i. Dually, $\&\{l_i : S_i\}_{1 \in I}$ describes a channel along which one of the labels l_i may be received, and which subsequently behaves as S_i. Finally, $\textsf{end}_!$ and $\textsf{end}_?$ describe channels that cannot be used for further communication. As we will see, it is convenient to use one if the last action on the channel is a send, and the other if the last action on the channel is a receive.

Types Types are defined by the following grammar:

$T, U, V ::=$	
S	session type (linear)
$T \otimes U$	tensor product (linear)
$T \multimap U$	function (linear)
$T \to U$	function (unlimited)
$\textsf{Unit}$	unit (unlimited)

Every session type is also a type, but not conversely. Types are formed from session types, tensor product, two forms of function space, and a unit for tensor product.

Each type is classified as linear or unlimited:

$$\textsf{lin}(S) \quad \textsf{lin}(T \otimes U) \quad \textsf{lin}(T \multimap U) \quad \textsf{un}(T \to U) \quad \textsf{un}(\textsf{Unit})$$

Here $\textsf{lin}(T)$ denotes a type that is linear, and $\textsf{un}(T)$ a type that is unlimited. Session types, tensor, and one type of function are limited; the other type of function and unit are unlimited. Unlimited types support weakening and contraction, while linear types do not. Unlimited types correspond to those written with ! in CP.

Duals Each session type S has a dual $\overline{S}$, defined as follows:

$$\begin{aligned} \overline{!T.S} &= ?T.\overline{S} \\ \overline{?T.S.} &= !T.\overline{S} \\ \overline{\oplus(l_i : S_i)_{i \in I}} &= \&(l_i : \overline{S}_i)_{i \in I} \\ \overline{\&(l_i : S_i)_{i \in I}} &= \oplus(l_i : \overline{S}_i)_{i \in I} \\ \overline{\textsf{end}_!} &= \textsf{end}_? \\ \overline{\textsf{end}_?} &= \textsf{end}_! \end{aligned}$$

Input is dual to output, selection is dual to choice, and the two terminators are dual. Duality between input and output does not take the dual of the type.

Duality is an involution, $\overline{\overline{S}} = S$.

Environments We let Φ, Ψ range over environments associating variables to types. Write $\textsf{un}(\Phi)$ to indicate that each type in Φ is unlimited. As in Section 2, order in environments is ignored and we use linear maintenance.

Terms Terms are defined by the following grammar:

$L, M, N ::=$	
x	identifier
unit	unit constant
$\lambda x.\, N$	function abstraction
$L\ M$	function application
(M, N)	pair construction
$\mathsf{let}\ (x, y) = M\ \mathsf{in}\ N$	pair deconstruction
$\mathsf{send}\ M\ N$	send value M on channel N
$\mathsf{receive}\ M$	receive from channel M
$\mathsf{select}\ l\ M$	select label l on channel M
$\mathsf{case}\ M\ \mathsf{of}\ \{l_i : x.N_i\}_{i \in I}$	offer choice on channel M
$\mathsf{with}\ x\ \mathsf{connect}\ M\ \mathsf{to}\ N$	connect M to N by channel x
$\mathsf{terminate}\ M$	terminate input

The first six operations specify a linear λ-calculus, and the remaining six specify communication along a channel.

The terms are best understood in conjunction with their type rules, shown in Figure 5. The rules for variables, unit, weakening, contraction, function abstraction and application, and pair construction and deconstruction are standard. Functions are either limited or unlimited. As usual, function abstraction may produce an unlimited function only if all of its free variables are of unlimited type. Following Gay and Vasconcelos (2010) we do not give a separate rule for application of unlimited function, but instead give a rule permitting an unlimited function to be treated as a linear function, which may then be applied using the rule for linear function application.

For simplicity, we do not require that each term have a unique type. In particular, a λ-expression where all free variables have unlimited type may be given either linear or unlimited function type. In a practical system, one might introduce subtyping and arrange that each term have a unique smallest type.

The rule for output is

$$\frac{\Phi \vdash M : T \qquad \Psi \vdash N : !T.S}{\Phi, \Psi \vdash \mathsf{send}\ M\ N : S}\ \textsf{Send}$$

Channels are managed linearly, so each operation on channels takes the channel before the operation as an argument, and returns the channel after the operation as the result. Executing ‘send M N’ outputs the value M of type T along channel N of session type $!T.S$, and returns the updated channel, which after the output has session type S.

The rule for input is

$$\frac{\Phi \vdash M : ?T.S}{\Phi \vdash \mathsf{receive}\ M : T \otimes S}\ \textsf{Receive}$$

Executing ‘receive M’ inputs a value from channel M of session type $?T.S$, and returns a pair consisting of the input value of type T, and the updated channel, which after the input has session type S. The returned pair must be linear because it contains a session type, which is linear.

Gay and Vasconcelos (2010) treat ‘send’ and ‘receive’ as function constants, and require two versions of ‘send’ to cope with complications arising from currying. We treat ‘send’ and ‘receive’ as language constructs, which avoids the need for two versions of ‘send’. Thanks to the rules for limited and unlimited function abstraction, $\lambda x.\, \lambda y.\, \mathsf{send}\ x\ y$ has type $T \multimap !T.S \multimap S$ and also type $T \to !T.S \multimap S$ when $\mathsf{un}(T)$.

The operations Select and Case are similar, and standard.

The rule to create new channels is:

$$\frac{\Phi, x : S \vdash M : \mathsf{end}_! \qquad \Psi, x : \overline{S} \vdash N : T}{\Phi, \Psi \vdash \mathsf{with}\ x\ \mathsf{connect}\ M\ \mathsf{to}\ N : T}\ \textsf{Connect}$$

Executing ‘with x connect M to N’ creates a new channel x with session type S, where x is used at type S within term M and at the dual type $\overline{S}$ within term N. The two terms M and N are evaluated concurrently. As is usual when forking off a value, only one of the two subterms returns a value that is passed to the rest of the program. The left subterm returns the exhausted channel, which has type $\mathsf{end}_!$. The right subterm returns a value of type T that is passed on to the rest of the program.

Finally, we require a rule to terminate the other channel:

$$\frac{\Phi \vdash M : T \otimes \mathsf{end}_?}{\Phi \vdash \mathsf{terminate}\ M : T}\ \textsf{Terminate}$$

Executing ‘terminate M’ evaluates term M, which returns a pair consisting of an exhausted channel of type $\mathsf{end}_?$ and a value of type T, then deallocates the channel and returns the value.

The constructs for Connect and Terminate between them deallocate two ends of a channel. The system is designed so it is convenient to use $\mathsf{end}_!$ on a channel whose last operation is Send, and $\mathsf{end}_?$ on a channel whose last operation is Receive.

Usually, session typed systems make end an unlimited type that is self-dual, but the formulation here fits better with CLL. A variation where end is a linear type requiring explicit deallocation is considered by Vasconcelos (2011).

One might consider alternative designs, say to replace Connect by an operation that creates a channel and returns both ends of it in a pair of type $S \otimes \overline{S}$, or to replace Terminate by an operation that takes a pair of type $\mathsf{end}_! \otimes \mathsf{end}_?$ and returns unit. However, both of these designs are difficult to translate into CP, which suggests they may suffer from deadlock.

3.1 Translation

The translation of GV into CP is given in Figures 6 and 7.

Session types The translation of session types is as follows:

$$\begin{aligned}
[\![!T.S]\!] &= [\![T]\!]^\perp \parr [\![S]\!] \\
[\![?T.S]\!] &= [\![T]\!] \otimes [\![S]\!] \\
[\![\oplus\{l_i : S_i\}_{i \in I}]\!] &= [\![S_1]\!] \mathbin{\&} \cdots \mathbin{\&} [\![S_n]\!], \quad I = \{1, \ldots, n\} \\
[\![\&\{l_i : S_i\}_{i \in I}]\!] &= [\![S_1]\!] \oplus \cdots \oplus [\![S_n]\!], \quad I = \{1, \ldots, n\} \\
[\![\mathsf{end}_!]\!] &= \perp \\
[\![\mathsf{end}_?]\!] &= 1
\end{aligned}$$

This translation is surprising, in that each operator translates to the dual of what one might expect! The session type for output in GV, $!T.S$ is translated into $\parr$, the connective that is interpreted as input in CP, and the session type for input in GV, $?T.S$ is translated into $\otimes$, the connective that is interpreted as output in CP. Similarly $\oplus$ and $\&$ in GV translate, respectively, to $\&$ and $\oplus$ in CP. Finally, $\mathsf{end}_!$ and $\mathsf{end}_?$ in GV translate, respectively, to $\perp$ and 1 in CP, the units for $\parr$ and $\otimes$.

The intuitive explanation of this duality is that Send and Receive in GV take channels as *arguments*, whereas the interpretation of the connectives in CP is for channels as *results*. Indeed, the send operation takes a value and a channel, and sends the value to that channel—in other words, the channel must *input* the value. Dually, the receive operation takes a channel and returns a value—in other words, the channel must *output* the value. A similar inversion occurs with respect to Select and Case.

Recall that duality on session types in GV leaves the types of sent and received values unchanged:

$$\overline{!T.S} = ?T.\overline{S} \qquad \overline{?T.S.} = !T.\overline{S}$$

Conversely, the translation of these operations takes the dual of the sent value, but not the received value:

$$[\![!T.S]\!] = [\![T]\!]^\perp \parr [\![S]\!] \qquad [\![?T.S]\!] = [\![T]\!] \otimes [\![S]\!]$$

In classical linear logic, $A \multimap B = A^\perp \parr B$, so the right-hand side of the first line could alternatively be written $[\![T]\!] \multimap [\![S]\!]$. Accordingly, and as one would hope, the translation preserves duality: $[\![\overline{S}]\!] = [\![S]\!]^\perp$.

$$\left[\!\!\left[\frac{}{x:T\vdash x:T}\,\mathsf{Id}\right]\!\!\right]z \;=\; \frac{}{x\leftrightarrow z\vdash x:[\![T]\!]^{\perp},\, z:[\![T]\!]}\,\mathsf{Ax} \qquad \left[\!\!\left[\frac{}{\vdash \mathsf{unit}:\mathsf{Unit}}\,\mathsf{Unit}\right]\!\!\right]z \;=\; \frac{\dfrac{}{y.\mathsf{case}()\vdash y:\top}\,\top}{!z(y).y.\mathsf{case}()\vdash z:!\top}\,!$$

$$\left[\!\!\left[\frac{\Phi\vdash N:U \quad \mathsf{un}(T)}{\Phi,\, x:T\vdash N:U}\,\mathsf{Weaken}\right]\!\!\right]z \;=\; \frac{[\![N]\!]z\vdash[\![\Phi]\!]^{\perp},\, z:[\![U]\!]}{[\![N]\!]z\vdash[\![\Phi]\!]^{\perp},\, x:[\![T]\!]^{\perp},\, z:[\![U]\!]}\,\mathsf{Weaken}$$

$$\left[\!\!\left[\frac{\Phi,\, x':T,\, x'':T\vdash N:U \quad \mathsf{un}(T)}{\Phi,\, x:T\vdash N\{x/x',x/x''\}:U}\,\mathsf{Contract}\right]\!\!\right]z \;=\; \frac{[\![N]\!]z\vdash[\![\Phi]\!]^{\perp},\, x':[\![T]\!]^{\perp},\, x'':[\![T]\!]^{\perp},\, z:[\![U]\!]}{[\![N\{x/x',x/x''\}]\!]z\vdash[\![\Phi]\!]^{\perp},\, x:[\![T]\!]^{\perp},\, z:[\![U]\!]}\,\mathsf{Contract}$$

$$\left[\!\!\left[\frac{\Phi,\, x:T\vdash N:U}{\Phi\vdash\lambda x.\,N:T\multimap U}\,\multimap\text{-}\mathsf{I}\right]\!\!\right]z \;=\; \frac{[\![N]\!]z\vdash[\![\Phi]\!]^{\perp},\, x:[\![T]\!]^{\perp},\, z:[\![U]\!]}{z(x).[\![N]\!]z\vdash[\![\Phi]\!]^{\perp},\, z:[\![T]\!]^{\perp}\parr[\![U]\!]}\,\parr$$

$$\left[\!\!\left[\frac{\Phi\vdash L:T\multimap U \quad \Psi\vdash M:T}{\Phi,\,\Psi\vdash L\,M:U}\,\multimap\text{-}\mathsf{E}\right]\!\!\right]z \;=\; \frac{[\![L]\!]y\vdash[\![\Phi]\!]^{\perp},\, y:[\![T]\!]^{\perp}\parr[\![U]\!] \quad \dfrac{[\![M]\!]x\vdash[\![\Psi]\!]^{\perp},\, x:[\![T]\!] \quad \dfrac{}{y\leftrightarrow z\vdash y:[\![U]\!]^{\perp},\, z:[\![U]\!]}\,\mathsf{Ax}}{y[x].([\![M]\!]x\mid y\leftrightarrow z)\vdash[\![\Psi]\!]^{\perp},\, y:[\![T]\!]\otimes[\![U]\!]^{\perp},\, z:[\![U]\!]}\,\otimes}{\nu y.([\![L]\!]y\mid y[x].([\![M]\!]x\mid y\leftrightarrow z))\vdash[\![\Phi]\!]^{\perp},\,[\![\Psi]\!]^{\perp},\, z:[\![U]\!]}\,\mathsf{Cut}$$

$$\left[\!\!\left[\frac{\Phi\vdash L:T\multimap U \quad \mathsf{un}(\Phi)}{\Phi\vdash L:T\to U}\,\to\text{-}\mathsf{I}\right]\!\!\right]z \;=\; \frac{[\![L]\!]y\vdash[\![\Phi]\!]^{\perp},\, y:[\![T\multimap U]\!]}{!z(y).[\![L]\!]y\vdash[\![\Phi]\!]^{\perp},\, z:![\![T\multimap U]\!]}\,!$$

$$\left[\!\!\left[\frac{\Phi\vdash L:T\to U}{\Phi\vdash L:T\multimap U}\,\to\text{-}\mathsf{E}\right]\!\!\right]z \;=\; \frac{[\![L]\!]y\vdash[\![\Phi]\!]^{\perp},\, y:![\![T\multimap U]\!] \quad \dfrac{\dfrac{}{x\leftrightarrow z\vdash x:[\![T\multimap U]\!]^{\perp},\, z:[\![T\multimap U]\!]}\,\mathsf{Ax}}{?y[x].x\leftrightarrow z\vdash y:?[\![T\multimap U]\!]^{\perp},\, z:[\![T\multimap U]\!]}\,?}{\nu y.([\![L]\!]y\mid ?y[x].x\leftrightarrow z)\vdash[\![\Phi]\!]^{\perp},\, z:[\![T\multimap U]\!]}\,\mathsf{Cut}$$

$$\left[\!\!\left[\frac{\Phi\vdash M:T \quad \Psi\vdash N:U}{\Phi,\,\Psi\vdash(M,N):T\otimes U}\,\otimes\text{-}\mathsf{I}\right]\!\!\right]z \;=\; \frac{[\![M]\!]y\vdash[\![\Phi]\!]^{\perp},\, y:[\![T]\!] \quad [\![N]\!]z\vdash[\![\Psi]\!]^{\perp},\, z:[\![U]\!]}{z[y].([\![M]\!]y\mid[\![N]\!]z)\vdash[\![\Phi]\!]^{\perp},\,[\![\Psi]\!]^{\perp},\, z:[\![T]\!]\otimes[\![U]\!]}\,\otimes$$

$$\left[\!\!\left[\frac{\Phi\vdash M:T\otimes U \quad \Psi,\, x:T,\, y:U\vdash N:V}{\Phi,\,\Psi\vdash \mathsf{let}\,(x,y)=M\ \mathsf{in}\ N:V}\,\otimes\text{-}\mathsf{E}\right]\!\!\right]z \;=\; \frac{[\![M]\!]y\vdash[\![\Phi]\!]^{\perp},\, y:[\![T]\!]\otimes[\![U]\!] \quad \dfrac{[\![N]\!]z\vdash[\![\Psi]\!]^{\perp},\, x:[\![T]\!]^{\perp},\, y:[\![U]\!]^{\perp},\, z:[\![V]\!]}{y(x).[\![N]\!]z\vdash[\![\Psi]\!]^{\perp},\, y:[\![T]\!]^{\perp}\parr[\![U]\!]^{\perp},\, z:[\![V]\!]}\,\parr}{\nu y.([\![M]\!]y\mid y(x).[\![N]\!]z)\vdash[\![\Phi]\!]^{\perp},\,[\![\Psi]\!]^{\perp},\, z:[\![V]\!]}\,\mathsf{Cut}$$

Figure 6. Translation from GV into CP, Part I

Types The translation of types is as follows:

$$\begin{aligned}[\![T\multimap U]\!] &= [\![T]\!]^{\perp}\parr[\![U]\!]\\ [\![T\to U]\!] &= !([\![T]\!]^{\perp}\parr[\![U]\!])\\ [\![T\otimes U]\!] &= [\![T]\!]\otimes[\![U]\!]\\ [\![\mathsf{Unit}]\!] &= !\top\end{aligned}$$

Session types are also types, they are translated as above.

The right-hand side of the first equation could alternatively be written $[\![T]\!]\multimap[\![U]\!]$, showing that linear functions translate as standard.

The right-hand side of the second equation could alternatively be written $!([\![T]\!]\multimap[\![U]\!])$. There are two standard translations of intuitionistic logic into classical linear logic or, equivalently, of λ-calculus into linear λ-calculus. Girard's original takes $(A\to B)^{\circ} = !A^{\circ}\multimap B^{\circ}$, and corresponds to call-by-name, while a lesser known alternative takes $(A\to B)^{*} = !(A^{*}\multimap B^{*})$, and correspond to call-by-value (see Benton and Wadler (1996) and Toninho et al. (2012)). The second is used here.

In classical linear logic, there is a bi-implication between 1 and $!\top$ (in many models, this bi-implication is an isomorphism), so the right-hand side of the last equation could alternatively be written 1, the unit for $\otimes$.

An unlimited type in GV translates to a type constructed with ! in CP: If $\mathsf{un}(T)$ then $[\![T]\!] = !A$, for some A.

Terms Translation of terms is written in a continuation-passing style standard for translations of λ-calculi into process calculi. The translation of term M of type T is written $[\![M]\!]z$ where z is a channel of type $[\![T]\!]$; the process that translates M transmits the answer it computes along z. More precisely, if $\Phi\vdash M:T$ then $[\![M]\!]z\vdash[\![\Phi]\!]^{\perp},\, z:[\![T]\!]$, where the Φ to the left of the turnstile in GV translates, as one might expect, to the dual $[\![\Phi]\!]^{\perp}$ on the right of the turn-style in CP.

The translation of terms is shown in Figures 6 and 7. Rather than simply giving a translation from terms of GV to terms of CP, we show the translation as taking type derivation trees to type derivation trees. Giving the translation on type derivation trees rather than terms has two advantages. First, it eliminates any ambiguity arising from the fact, noted previously, that terms in GV do not have unique types. Second, it makes it easy to validate that the translation preserves types.

Figure 6 shows the translations for operations of a linear λ-calculus. A variable translates to an axiom, weakening and contraction translate to weakening and contraction. Function abstraction and product deconstruction both translate to input, and func-

$$\left[\!\!\left[\frac{\Phi \vdash M : T \quad \Psi \vdash N : !T.S}{\Phi, \Psi \vdash \mathsf{send}\ M\ N : S}\ \mathsf{Send}\right]\!\!\right] z \quad =$$

$$\dfrac{\dfrac{[\![M]\!]y \vdash [\![\Phi]\!]^{\perp}, y : [\![T]\!] \qquad \dfrac{}{x \leftrightarrow z \vdash x : [\![S]\!]^{\perp}, z : [\![S]\!]}\,\mathsf{Ax}}{x[y].([\![M]\!]y \mid x \leftrightarrow z) \vdash [\![\Phi]\!]^{\perp}, x : [\![T]\!] \otimes [\![S]\!]^{\perp}}\,\otimes \qquad [\![N]\!]x \vdash [\![\Psi]\!]^{\perp}, x : [\![T]\!]^{\perp} \parr [\![S]\!]}{\nu x.(x[y].([\![M]\!]y \mid x \leftrightarrow z) \mid [\![N]\!]x) \vdash [\![\Phi]\!]^{\perp}, [\![\Psi]\!]^{\perp}, z : [\![S]\!]}\,\mathsf{Cut}$$

$$\left[\!\!\left[\frac{\Phi \vdash M : ?T.S}{\Phi \vdash \mathsf{receive}\ M : T \otimes S}\ \mathsf{Receive}\right]\!\!\right] z \quad = \quad [\![M]\!]z \vdash [\![\Phi]\!], z : [\![T]\!] \otimes [\![S]\!]$$

$$\left[\!\!\left[\frac{\Phi \vdash M : \oplus\{l_i : S_i\}_{i \in I}}{\Phi \vdash \mathsf{select}\ l_j\ M : S_j}\ \mathsf{Select}\right]\!\!\right] z \quad =$$

$$\dfrac{[\![M]\!]x \vdash [\![\Phi]\!]^{\perp}, x : [\![S_1]\!] \mathbin{\&} \cdots \mathbin{\&} [\![S_n]\!] \qquad \dfrac{\dfrac{}{x \leftrightarrow z \vdash x : [\![S_j]\!]^{\perp}, z : [\![S_j]\!]}\,\mathsf{Ax}}{x[\mathrm{in}_j].x \leftrightarrow z \vdash x : [\![S_1]\!]^{\perp} \oplus \cdots \oplus [\![S_n]\!]^{\perp}, z : [\![S_j]\!]}\,\oplus_{\mathsf{i}}}{\nu x.([\![M]\!]x \mid x[\mathrm{in}_j].x \leftrightarrow z) \vdash [\![\Phi]\!]^{\perp}, z : [\![S_j]\!]}\,\mathsf{Cut}$$

$$\left[\!\!\left[\frac{\Phi \vdash M : \&\{l_i : S_i\}_{i \in I} \quad (\Psi, x : S_i \vdash N_i : T)_{i \in I}}{\Phi, \Psi \vdash \mathsf{case}\ M\ \mathsf{of}\ \{l_i : x.N_i\}_{i \in I} : T}\ \mathsf{Case}\right]\!\!\right] z \quad =$$

$$\dfrac{[\![M]\!]x \vdash [\![\Phi]\!]^{\perp}, x : [\![S_1]\!] \oplus \cdots \oplus [\![S_n]\!] \qquad \dfrac{([\![N_i]\!]z \vdash [\![\Psi]\!]^{\perp}, x : [\![S_i]\!]^{\perp}, z : [\![T]\!])_{i \in I}}{x.\mathsf{case}([\![N_1]\!], \ldots, [\![N_n]\!]) \vdash x : [\![S_1]\!] \mathbin{\&} \cdots \mathbin{\&} [\![S_n]\!], z : [\![T]\!]}\,\&}{\nu x.([\![M]\!]x \mid x.\mathsf{case}([\![N_1]\!], \ldots, [\![N_n]\!])) \vdash [\![\Phi]\!]^{\perp}, [\![\Psi]\!]^{\perp}, z : [\![T]\!]}\,\mathsf{Cut}$$

$$\left[\!\!\left[\frac{\Phi, x : S \vdash M : \mathsf{end}_{!} \quad \Psi, x : \overline{S} \vdash N : T}{\Phi, \Psi \vdash \mathsf{with}\ x\ \mathsf{connect}\ M\ \mathsf{to}\ N : T}\ \mathsf{Connect}\right]\!\!\right] z \quad =$$

$$\dfrac{\dfrac{[\![M]\!]y \vdash [\![\Phi]\!]^{\perp}, x : [\![S]\!]^{\perp}, y : \perp \qquad \dfrac{}{y[\,].0 \vdash y : 1}\,1}{\nu y.([\![M]\!]y \mid y[\,].0) \vdash [\![\Phi]\!]^{\perp}, x : [\![S]\!]^{\perp}}\,\mathsf{Cut} \qquad [\![N]\!]z \vdash [\![\Psi]\!]^{\perp}, x : [\![S]\!], z : [\![T]\!]}{\nu x.(\nu y.([\![M]\!]y \mid y[\,].0) \mid [\![N]\!]z) \vdash [\![\Phi]\!]^{\perp}, [\![\Psi]\!]^{\perp}, z : [\![T]\!]}\,\mathsf{Cut}$$

$$\left[\!\!\left[\frac{\Phi \vdash M : T \otimes \mathsf{end}_{?}}{\Phi \vdash \mathsf{terminate}\ M : T}\ \mathsf{Terminate}\right]\!\!\right] z \quad = \quad \dfrac{[\![M]\!]y \vdash [\![\Phi]\!]^{\perp}, y : [\![T]\!] \otimes 1 \qquad \dfrac{\dfrac{\dfrac{}{z \leftrightarrow y \vdash z : [\![T]\!], y : [\![T]\!]^{\perp}}\,\mathsf{Ax}}{x().z \leftrightarrow y \vdash z : [\![T]\!], y : [\![T]\!]^{\perp}, x : \perp}\,\perp}{y(x).x().z \leftrightarrow y \vdash z : [\![T]\!], y : [\![T]\!]^{\perp} \parr \perp}\,\parr}{\nu y.([\![M]\!]y \mid y(x).x().z \leftrightarrow y) \vdash [\![\Phi]\!]^{\perp}, z : [\![T]\!]}\,\mathsf{Cut}$$

Figure 7. Translation from GV into CP, Part II

tion application and product construction both translate to output. The translation of each elimination rule ($\multimap$-E, $\to$-E, and $\otimes$-E) also requires a use of Cut.

Figure 7 shows the translation for operations for communication. For purposes of the translation, it is convenient to work with n-fold analogues of $\oplus$ and $\&$, writing $\in_i$ for selection and $\mathsf{case}(P_1, \cdots, P_n)$ for choice.

Despite the inversion noted earlier in the translation of session types, the translation of Send involves an output operation of the form $x[y].(P \mid Q)$, the translation of Select involves an select operation of the form $x[\mathrm{in}_j].P$, the translation of Case involves a choice operation of the form $\mathsf{case}(Q_1, \ldots, Q_n)$, the translation of $\mathsf{end}_{!}$ in Connect involves an empty output of the form $y[\,].0$, and the translation of Terminate involves an empty input of the form $x().P$. Each of these translations also introduces a Cut, corresponding to communication with supplied channel. The translation of Receive is entirely trivial, but the corresponding input operation of the form $x(y).R$ appears in the translation of $\otimes$-E, which deconstructs the returned pair. Finally, the translation of Connect involves a Cut, which corresponds to introducing a channel for communication between the two subterms.

The translation preserves types.

THEOREM 3. *If* $\Phi \vdash M : T$ *then* $[\![M]\!]x \vdash [\![\Phi]\!]^{\perp}, x : [\![T]\!]$.

Proof sketch. See Figures 6 and 7. □

We also claim that the translation preserves the intended semantics. The formal semantics of Gay and Vasconcelos (2010) is based on asynchronous buffered communication, which adds additional complications, so we leave a formal proof of correspondence between the two for future work.

4. Related work

Session types Session types were introduced by Honda (1993), and further extended by Takeuchi et al. (1994), Honda et al. (1998), and Yoshida and Vasconcelos (2007). Subtyping for session types is considered by Gay and Hole (2005), and the linear functional language for session types considered in this paper was introduced by Gay and Vasconcelos (2010). Session types have been applied to describe operating system services by Fähndrich et al. (2006).

Deadlock freedom Variations on session types that guarantees deadlock freedom are presented in Sumii and Kobayashi (1998) and Carbone and Debois (2010). Unlike CP, where freedom from deadlock follows from the relation to cut elimination, in the first it is ensured by introducing a separate partial order on time tags, and in the second by introducing a constraint on underlying dependency graphs.

Linear types for process calculus A variety of linear types systems for process calculus are surveyed by Kobayashi (2002). Most of these systems look rather different than session types, but Kobayashi et al. (1996) presents an embedding of session types into a variant of π-calculus with linear types for channels.

Linear proof search Functional programming can be taken as arising from the Curry-Howard correspondence, by associating program evaluation with proof normalisation. Analogously, logic programming can be taken as arising by associating program evaluation with proof search. Logic programming approaches based on linear logic give rise to systems with some similarities to CP, see Miller (1992) and Kobayashi and Yonezawa (1993, 1994, 1995).

Polymorphism CP's support of polymorphism is based on the polymorphic π-calculus introduced by Turner (1995) and further discussed by Pierce and Turner (2000) and Pierce and Sangiorgi (2000), which uses explict polymorphism (Church-style). In contrast, Berger et al. (2005) introduce a polymorphically typed session calculus that uses implicit polymorphism (Curry-style).

Linear logic as a process calculus Various interpretations of linear logic as a process calculus are proposed by Abramsky (1993), Abramsky (1994), and Abramsky et al. (1996), the second of these being elaborated in detail by Bellin and Scott (1994).

This paper is inspired by a series of papers by Caires, Pfenning, Toninho, and Pérez. Caires and Pfenning (2010) first observed the correspondence relating formulas of linear logic to session types; its journal version is Caires et al. (2012b). Pfenning et al. (2011) extends the correspondence to dependent types in a stratified system, with concurrent communication at the outer level and a dependently-typed functional language at the inner level, and Pfenning et al. (2011) extends further to support proof-carrying code and proof irrelevance. Toninho et al. (2012) explores encodings of λ-calculus into πDILL. Pérez et al. (2012) introduces logical relations on linear-typed processes to prove termination and contextual equivalences. Caires et al. (2012a) is the text of an invited talk at TLDI, summarising the above.

Mazurak and Zdancewic (2010) present Lolliproc, which also offers a Curry-Howard interpretation of session types by relating the call/cc control operators to communication using a double-negation operator on types.

DILL vs. CLL Caires et al. (2012b) consider a variant of πDILL based on one-sided sequents of classical linear logic, which they call πCLL. Their πCLL is similar to CP, but differs in important particulars: its bookkeeping is more elaborate, using two zones, one linear and one intuitionitic; it has no axiom, so cannot easily support polymorphism; and it does not support reductions corresponding to the commuting conversions.

Caires et al. (2012b) state they prefer a formulation based on DILL to one based on CLL, because DILL satisfies a locality property for replicated input while CLL does not. Locality requires that names received along a channel may be used to send output but not to receive input, and is useful both from an implementation point of view and because a process calculus so restricted satisfies additional observational equivalences, as shown by Merro and Sangiorgi (2004). Caires et al. (2012b) only restrict replicated input, because restricting *all* input is too severe for a session-typed calculus. However, the good properties of locality have been studied only in the case where *all* input is prohibited on received names. It remains to be seen to what extent the fact that DILL imposes locality for replicated names is significant.

Additionally, in a private conversation, Pfenning relayed that he believes DILL may be amenable to extension to dependent types, while he suspects CLL is not because strong sums become degenerate in some classical settings, as shown by Herbelin (2005). However, linear logic is more amenable to constructive treatment than traditional classical logic, as argued by Girard (1991), so it remains unclear to what extent CP, or πCLL, may support dependent types.

5. Conclusion

One reason λ-calculus provides such a successful foundation for functional programming is that it includes both fragments that guarantee termination (typed λ-calculi) and fragments that can model any recursive function (untyped λ-calculus, or typed λ-calculi augmented with a general fixpoint operator). Indeed, the former can be seen as giving rise to the latter, by considering recursive types with recursion in negative positions; untyped λ-calculus can be modelled by a solution to the recursion equation $X \simeq X \to X$. Similarly, a foundation for concurrency based on linear logic will be of limited value if it only models deadlock-free processes. Are there extensions that support more general forms of concurrency?

Girard (1987) proposes one such extension, the Mix rule. In our notation, this is written:

$$\frac{P \vdash \Gamma \quad Q \vdash \Delta}{P \mid Q \vdash \Gamma, \Delta}\ \textsf{Mix}$$

Mix differs from Cut in that there are *no* channels in common between P and Q, rather than one. Mix is equivalent to provability of the proposition $A \otimes B \multimap A \parr B$. Systems with Mix still do not deadlock, but support concurrent structures that cannot arise under CLL, namely, systems with two components that are independent. Caires et al. (2012a) consider two variations of the rules for 1 and $\bot$, the second of which is less restrictive and derives a rule similar to Mix.

Abramsky et al. (1996) proposes another extension, the Binary Cut rule (a special case of Multicut). In our notation, this is written:

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

Binary Cut differs from Cut in that there are *two* channels in common between P and Q, rather than one. Binary Cut is equivalent to provability of the proposition $A \parr B \multimap A \otimes B$. Binary Cut allows one to express systems where communications form a loop and may deadlock.

Systems with both Mix and Binary Cut are compact, in that from either of $A \otimes B$ and $A \parr B$ one may derive the other. Abramsky et al. (1996) provides a translation of full π-calculus into a compact linear system, roughly analogous to the embedding of untyped λ-calculus into typed λ-calculus based on the isomorphism $X \simeq X \to X$.

Searching for principled extensions of CP that support the unfettered power of the full π-calculus is a topic for future work. As λ-calculus provided foundations for functional programming in the last century, may we hope for this emerging calculus to provide foundations for concurrent programming in the coming century?

Acknowledgements For comments and discussions, my thanks to Samson Abramsky, Luis Caires, Marco Gaboardi, Simon Gay, Andy Gordon, Marc Hamann, Jiansen He, Kohei Honda, Luke Ong, Frank Pfenning, Colin Stirling, Michael Stone, Vasco Vasconcelos, Hongseok Yang, Nobuko Yoshida, Stephan Zdancewic, and the anonymous referees.

References

Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1&2):3–57, 1993.

Samson Abramsky. Proofs as processes. *Theoretical Computer Science*, 135(1):5–9, 1994.

Samson Abramsky, Simon J. Gay, and Rajagopal Nagarajan. Interaction categories and the foundations of typed concurrent programming. In *Deductive Program Design, Marktoberdorf*, pages 35–113, 1996.

Gianluigi Bellin and Philip J. Scott. On the pi-calculus and linear logic. *Theoretical Computer Science*, 135(1):11–65, 1994.

Nick Benton and Philip Wadler. Linear logic, monads and the lambda calculus. In *Logic in Computer Science (LICS)*, pages 420–431, 1996.

Martin Berger, Kohei Honda, and Nobuko Yoshida. Genericity and the pi-calculus. *Acta Inf.*, 42(2-3):83–141, 2005.

Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, pages 222–236, 2010.

Luís Caires, Frank Pfenning, and Bernardo Toninho. Towards concurrent type theory. In *Types in Language Design and Implementation (TLDI)*, January 2012a.

Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 2012b. Submitted.

Marco Carbone and Søren Debois. A graphical approach to progress for structured communication in web services. In *Interaction and Concurrency Experience (ICE)*, pages 13–27, 2010.

Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *European Conference on Computer Systems (EuroSys)*, pages 177–190, 2006.

Jean H. Gallier. *On Girard's "Candidats de Reducibilité"*, pages 123–204. Academic Press, 1990.

Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005.

Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50: 1–102, 1987.

Jean-Yves Girard. A new constructive logic: Classical logic. *Mathematical Structures in Computer Science*, 1(3):255–296, 1991.

Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

Hugo Herbelin. On the degeneracy of sigma-types in presence of computational classical logic. In *Typed Lambda Calculi and Applications (TLCA)*, pages 209–220, 2005.

Kohei Honda. Types for dyadic interaction. In *CONCUR*, pages 509–523, 1993.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming (ESOP)*, pages 122–138, 1998.

Naoki Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, LNCS 2757, pages 439–453, 2002.

Naoki Kobayashi and Akinori Yonezawa. Acl — a concurrent linear logic programming paradigm. In *International Logic Programming Symposium (ILPS)*, pages 279–294, 1993.

Naoki Kobayashi and Akinori Yonezawa. Higher-order concurrent linear logic programming. In *Theory and Practice of Parallel Programming*, LNCS 907, pages 137–166, 1994.

Naoki Kobayashi and Akinori Yonezawa. Asynchronous communication model based on linear logic. *Formal Aspects of Computing*, 7(2):113–149, 1995.

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *POPL*, pages 358–371, 1996.

Karl Mazurak and Steve Zdancewic. Lolliproc: to concurrency from classical linear logic via curry-howard and control. In *International Conference on Functional Programming (ICFP)*, pages 39–50, 2010.

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

Dale Miller. The pi-calculus as a theory in linear logic: Preliminary results. In *Extensions to Logic Programming*, LNCS 660, pages 242–264, 1992.

Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, 1992.

Jorge Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Termination in session-based concurrency via linear logical relations. In *European Symposium on Programming (ESOP)*, 2012.

Frank Pfenning, Luís Caires, and Bernardo Toninho. Proof-carrying code in a session-typed process calculus. In *Certified Programs and Proofs (CPP)*, pages 21–36, 2011.

Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM*, 47(3):531–584, 2000.

Benjamin C. Pierce and David N. Turner. Pict: a programming language based on the pi-calculus. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 455–494. The MIT Press, 2000. ISBN 978-0-262-16188-6.

Andrew M. Pitts. Structural recursion with locally scoped names. *J. Funct. Program.*, 21(3):235–286, 2011.

Eijiro Sumii and Naoki Kobayashi. A generalized deadlock-free process calculus. In *High-Level Concurrent Languages (HLCL)*, 1998. *ENTCS* 16(3):225–247, 1998.

Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In C. Halatsis, D. G. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE*, LNCS 817, pages 398–413, 1994.

Bernardo Toninho, Luís Caires, and Frank Pfenning. Functions as session-typed processes. In *Foundations of Software Science and Computation (FoSSaCS)*, 2012.

David N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995.

Vasco T. Vasconcelos. Sessions, from types to programming languages. *Bulletin of the European Association for Theoretical Computer Science*, 103:53–73, 2011.

Nobuko Yoshida and Vasco T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electronic Notes in Theoretical Computer Science*, 171(4):73–93, 2007.