

# A Semantics for Propositions as Sessions

Sam Lindley and J. Garrett Morris

The University of Edinburgh  
{Sam.Lindley, Garrett.Morris}@ed.ac.uk

**Abstract.** Session types provide a static guarantee that concurrent programs respect communication protocols. Recently Caires, Pfenning, and Toninho, and Wadler, have developed a correspondence between the propositions of linear logic and session typed  $\pi$ -calculus processes. We relate the cut-elimination semantics of this approach to an operational semantics for session-typed concurrency in a functional language. We begin by presenting a variant of Wadler’s minimal session-typed functional language, GV. We give a small-step operational semantics for GV. We develop a suitable notion of deadlock for our functional setting, based on existing approaches for capturing deadlock in  $\pi$ -calculus, and show that all well-typed GV programs are deadlock-free, deterministic, and terminating. We relate GV to linear logic by giving translations between GV and CP, a process calculus with a type system and semantics based on classical linear logic. We prove that both directions of our translation preserve reduction; previous translations from GV to CP, in contrast, failed to preserve  $\beta$ -reduction. Crucially, we use a weak form of explicit substitutions, effectively generalising lambda abstractions to closures, in order to maintain a correspondence with cut reduction. Finally, to demonstrate the modularity of our approach, we define two extensions of GV and show that they preserve deadlock freedom.

## 1 Introduction

From massively distributed programs running across entire data centres, to hand-held apps reliant on remote services for functionality, concurrency has become a critical aspect of modern programs and thus a central problem in program correctness. Assuring correct concurrent behaviour requires reasoning not just about the types of data communicated, but the order in which the communication takes place. For example, the messages between an SMTP client and server are all strings, but a client that sends the recipient’s address before the sender’s address is in violation of the protocol, despite sending the correct type of data.

Session types, originally proposed by Honda [10], provide a mechanism to reason about the state of communications. The type of a session captures the expected behavior of a process communicating on that session. For example, we could express a simplified session type for an SMTP client as

$$!SenderAddress.!RcptAddress.!Message.end,$$

where  $!T.S$  denotes the expectation that the process will send a value of type  $T$ , then continue with the behavior specified by  $S$ . An important property of

session types is duality: we can derive the type of an SMTP server session from the client’s session type:

$$?SenderAddress.?RcptAddress.?Message.end,$$

where  $?T.S$  denotes the expectation that the process will send a value of type  $T$ , then continue with the behavior specified by  $S$ . Honda originally defined session types for process calculi; recent work [8, 21] has investigated the use of session types for concurrency in functional languages.

Session type systems are necessarily substructural—if processes can freely discard or duplicate sessions, then the type system cannot guarantee that the observable messages on sessions match the expectations of their types. Recent work has sought to establish a correspondence between session types and linear logic, the archetypal substructural logic for reasoning about state. Caires and Pfenning [4] developed a correspondence between cut elimination in intuitionistic linear logic and process reduction in a session-typed process calculus. Wadler [22] adapted their approach to classical linear logic, emphasising the role of duality in typing; the semantics of his system is given directly by the cut elimination rules of classical linear logic. He also gives a type-preserving translation from a simple functional calculus (GV) to a process calculus (CP); however, he gives no semantics for GV besides the semantics of the translated terms.

In this paper, we develop a minimal, session-typed functional language, also called GV. (Our language shares most of the distinctive features of Wadler’s, although it differs in some details.) We present a small-step operational semantics for GV, factored into functional and concurrent portions following the approach of Gay and Vasconcelos [8]. The functional portion of our semantics differs from standard presentations of call-by-value reduction only in that we adopt a weak form of explicit substitution to better maintain a connection with cut reduction. The concurrent portion of our semantics includes the typical reductions and equivalences of  $\pi$ -calculus-like systems. We develop a characterisation of deadlock-free programs in our setting; this is not completely trivial because closed expressions may return channels, so we cannot simply require that all possible communication take place. We show that well-typed GV programs are deadlock-free, deterministic, and terminating. We give translations between our variant of GV and Wadler’s CP, and show that these translations preserve reduction as well as typing. Finally, we show that GV is a useful basis for further language development by giving two extensions of GV and showing that our deadlock freedom results transfer easily to the extended languages.

The paper proceeds as follows.

- We define a core linearly-typed functional language, by extending linear lambda calculus with session-typed communication primitives (§2.1). We present an (untyped) synchronous operational semantics for our core language (§2.2). We characterise deadlock and normal forms; we show that typed terms are deadlock-free, that closed typed terms evaluate to normal forms (§2.3), and that evaluation is deterministic and terminating (§2.4).

Session types	$S ::= !T.S \mid ?T.S \mid \text{end}_? \mid \text{end}_! \mid S^\sharp$
Types	$T, U ::= S \mid \mathbf{1} \mid T \times U \mid \mathbf{0} \mid T + U \mid T \multimap U$
Terms	$L, M, N ::= x \mid K M \mid \lambda x.M \mid M N$ $\mid (M, N) \mid \text{let } (x, y) = M \text{ in } N$ $\mid \text{inl } M \mid \text{inr } M \mid \text{case } M \{ \text{inl } x \mapsto N; \text{inr } x \mapsto N \}$ $\mid () \mid M; N \mid \text{absurd } M$
Constants	$K ::= \text{send} \mid \text{receive} \mid \text{fork} \mid \text{wait} \mid \text{link}$

**Fig. 1:** Syntax of GV Terms and Types

- We connect our language to the interpretation of session types as linear logic propositions, by establishing a correspondence between the semantics of our language and that of CP. We begin by introducing CP (§3). We show that we can simulate CP reduction in GV (§3.1), and GV reduction in CP (§3.2). As  $\pi$ -calculus-like process calculi provide substitution only for names, not entire process expressions, the latter depends crucially on the use of weak explicit substitutions in the semantics of GV lambda abstractions.
- We consider two extensions of our core language: one which has a single, self-dual type for closed sessions, harmonising the treatment of closed channels with their treatment in other session-typed calculi (§4.1), and another which adds unlimited types and replicated behavior (§4.2). We show that these extensions preserve the essential deadlock freedom properties of the core language.

We conclude by discussing related (§5) and future (§6) work.

## 2 A Session-Typed Functional Language

### 2.1 Syntax and Typing

Figure 1 gives the syntax of GV types and terms. The types  $T$  include nullary ( $\mathbf{0}$ ) and binary ( $T + U$ ) linear sums, nullary ( $\mathbf{1}$ ) and binary ( $T \times U$ ) linear products, and linear implication ( $T \multimap U$ ). We frequently write  $M; N$  as the elimination form of  $\mathbf{1}$  in place of the more verbose  $\text{let } () = M \text{ in } N$ . Session types  $S$  include input ( $?T.S$ ), output ( $!T.S$ ), and closed sessions ( $\text{end}_?$ ,  $\text{end}_!$ ). We also include a type  $S^\sharp$  of channels; values of channel type cannot be used directly in terms, but will appear in the typing of thread configurations. The terms are the standard  $\lambda$ -calculus terms, augmented with constructs for pairs and sums. Figure 2 gives both typing rules and type schemas for the constants.

**Concurrency.** Concurrent behavior is provided by the constants. Communication is provided by **send** and **receive**. For example (assuming an extension of our core language with numbers and arithmetic operators), a computation  $M$  that received a pair of numbers along a channel  $z$  and then sent their sum along the same channel could be expressed by

$$M \triangleq \text{let } ((x, y), z) = \text{receive } z \text{ in send } (x + y, z)$$

Typing rules		
$\frac{T \neq S^\sharp}{\{x : T\} \vdash x : T}$	$\frac{K : T \multimap U \quad \Gamma \vdash M : T}{\Gamma \vdash K M : U}$	
$\frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x. M : T \multimap U}$	$\frac{\Gamma \vdash M : T \multimap U \quad \Gamma' \vdash N : T}{\Gamma, \Gamma' \vdash M N : U}$	
$\frac{\Gamma \vdash M : T \quad \Gamma' \vdash N : U}{\Gamma, \Gamma' \vdash (M, N) : T \times U}$	$\frac{\Gamma \vdash M : T \times T' \quad \Gamma', x : T, y : T' \vdash N : U}{\Gamma, \Gamma' \vdash \text{let } (x, y) = M \text{ in } N : U}$	
$\frac{\Gamma \vdash M : T}{\Gamma \vdash \text{inl } M : T + U}$	$\frac{\Gamma \vdash M : T + T' \quad \Gamma', x : T \vdash N : U \quad \Gamma', x : T' \vdash N' : U}{\Gamma, \Gamma' \vdash \text{case } M \{ \text{inl } x \mapsto N; \text{inr } x \mapsto N' \} : U}$	
$\frac{}{\vdash () : \mathbf{1}}$	$\frac{\Gamma \vdash M : \mathbf{1} \quad \Gamma' \vdash N : T}{\Gamma, \Gamma' \vdash \text{let } () = M \text{ in } N : T}$	$\frac{\Gamma \vdash M : \mathbf{0}}{\Gamma, \Gamma' \vdash \text{absurd } M : T}$
Type schemas for constants		
$\text{send} : T \times !T.S \multimap S$	$\text{receive} : ?T.S \multimap T \times S$	$\text{fork} : (S \multimap \text{end}_!) \multimap \overline{S}$
$\text{wait} : \text{end}_? \multimap \mathbf{1}$	$\text{link} : S \times \overline{S} \multimap \text{end}_!$	
Duality		
$\overline{!T.S} = ?T.\overline{S}$	$\overline{?T.S} = !T.\overline{S}$	$\overline{\text{end}_?} = \text{end}_! \quad \overline{\text{end}_!} = \text{end}_?$

**Fig. 2:** GV Typing Rules

(where the interpretation of nested patterns by sequences of bindings is standard). Sessions are treated linearly in GV. Thus, **receive** returns not only the received value (the pair of  $x$  and  $y$ ) but also a new copy of the session used for receiving ( $z$ ); similarly, **send** returns a copy of the session used for sending. Thus, the term above is well-typed in the context  $z : ?(Int \times Int).!Int.S$ , and evaluates to a session of type  $S$ . Session initiation is provided by **fork**. If  $f$  is a function from a session of type  $S$  to a closed session (of type  $\text{end}_!$ ), then **fork**  $f$  forks a new thread in which  $f$  is applied to a fresh session of type  $S$ , and returns a session of type  $\overline{S}$  in order to communicate with the thread. For example, the term **fork**  $(\lambda z.M)$  returns a channel of type  $!(Int \times Int).?Int.\text{end}_?$ . Given a thread created by **fork**  $f$ , the session returned from  $f$  is closed by **fork**; however, the other end of the channel must be closed by calling **wait**. A client of the process  $M$  could be defined as follows:

$$N \triangleq \text{let } z = \text{send } ((6, 7), z) \text{ in let } (x, z) = \text{receive } z \text{ in wait } z; x$$

The combined process **let**  $x = \text{fork}(\lambda z.M)$  **in**  $N$  evaluates to 13. The form **link**  $(x, y)$  forwards messages sent on  $x$  to be received on  $y$  and vice versa.

**Choice.** In addition to input and output, typical session type systems also provide session types representing internal ( $S_1 \oplus S_2$ ) and external ( $S_1 \& S_2$ ) choice (also known as selection and branching, respectively). For example, we could write a process that can either sum two numbers or negate one:

$$\text{offer } z \{ \text{inl } z \mapsto \text{let } ((x, y), z) = \text{receive } z \text{ in send } (x + y, z) \\ \text{inr } z \mapsto \text{let } (x, z) = \text{receive } z \text{ in send } (-x, z) \}$$

This term initially requires  $z : (? (Int \times Int) . !Int.S) \& (?Int. !Int.S)$ . A client of this process would begin by choosing which branch of the session to take; for example, we could extend the preceding example as follows:

$$\text{let } z = \text{select inl } z \text{ in let } z = \text{send } ((6, 7), z) \text{ in let } (x, z) = \text{receive } z \text{ in wait } z; x$$

While we would expect a surface language to include selection and branching, we omit them from our core calculus. Instead, we show that they are macro-expressible using the linear sum type. The intuition is that selection is implemented by sending a suitably tagged process, while branching is implemented by a term-level branch on a received value. Concretely, we define the types by:

$$S_1 \& S_2 \triangleq ?(S_1 + S_2).\text{end}_? \quad S_1 \oplus S_2 \triangleq !(S_1 + S_2).\text{end}_!$$

Note that we have the expected duality relationship:  $\overline{S_1 \& S_2} = \overline{S_1} \oplus \overline{S_2}$ . We can implement the **select** and **offer** primitives as follows (where  $\ell$  ranges over  $\{\text{inl}, \text{inr}\}$ ):

$$\text{select } \ell M \triangleq \text{fork}(\lambda x. \text{send } (\ell x, M)) \\ \text{offer } M \{ \text{inl } x \mapsto P; \text{inr } x \mapsto Q \} \triangleq \text{let } (x, y) = \text{receive } M \text{ in} \\ \text{wait } y; \text{case } x \{ \text{inl } x \mapsto P; \text{inr } x \mapsto Q \}$$

Correspondingly, nullary choice and selection may be encoded with the **0** type:

$$\&\{\} \triangleq ?\mathbf{0}.\text{end}_? \quad \oplus \{\} \triangleq !\mathbf{0}.\text{end}_! \\ \text{offer } M \{\} \triangleq \text{let } (x, y) = \text{receive } M \text{ in wait } y; \text{absurd}\{\}$$

## 2.2 Evaluation

Following Gay and Vasconcelos [8], we factor the semantics of GV into a (deterministic) reduction relation on terms (called  $\longrightarrow_V$ ) and a (non-deterministic) reduction on configurations of processes (called  $\longrightarrow$ ). Figure 3 gives the syntax of values, configurations, and evaluation and configuration contexts.

**Terms.** To preserve a close connection between the semantics of our term language and cut-reduction in linear logic, we define term reduction using weak explicit substitutions [15]. In this approach, we capture substitutions at  $\lambda$ -terms rather than immediately applying them to the body of the term. Our language

Values	$V, W ::= x \mid \lambda^\sigma x.M$ $\mid () \mid (V, W) \mid \text{inl } V \mid \text{inr } V$
Substitutions	$\sigma ::= \{V_1/x_1, \dots, V_n/x_n\}$ where the $x_i$ are pairwise distinct
Evaluation contexts	$E ::= [] \mid E M \mid V E \mid K V \mid E; M$ $\mid (E, M) \mid (V, E) \mid \text{let } (x, y) = E \text{ in } M$ $\mid \text{inl } E \mid \text{inr } E \mid \text{case } E \{ \text{inl } x \mapsto N; \text{inr } x \mapsto N' \}$ $F ::= \phi E$
Configurations	$C, D ::= \phi M \mid C \parallel C' \mid (\nu x)C$
Configuration contexts	$G ::= [] \mid G \parallel P \mid (\nu x)G$
Flags	$\phi ::= \circ \mid \bullet$

**Fig. 3:** Syntax of Values, Configurations, and Contexts

of terms, therefore, includes closures  $\lambda^\sigma x.M$ , where  $\sigma$  provides the captured substitution. We extend the typing judgement to include closures, as follows:

$$\frac{\Gamma, x : T \vdash M\sigma : U \quad \text{dom}(\sigma) = (\text{fv}(M) \setminus \{x\})}{\Gamma \vdash \lambda^\sigma x.M : T \multimap U}$$

The free variables of a closure  $\lambda^\sigma x.M$  are the free variables of the range of  $\sigma$ , not the free variables of  $M$ . The capture avoiding substitution  $M\sigma$  of  $\sigma$  applied to  $M$  is defined as usual on the free variables of  $M$ . Note that the side condition on the domain of  $\sigma$  is preserved under substitution. We implicitly treat plain lambda abstractions  $\lambda x.M$  as closures  $\lambda^\sigma x.M$ , where  $\sigma$  is a renaming substitution restricted to the free variables of  $M$  less  $\{x\}$ ; concretely:

$$\begin{aligned} \lambda x.M &\triangleq \lambda^\sigma x.(M\sigma') \\ \text{where } \text{fv}(M) &= \{x_1, \dots, x_n\} & y_1, \dots, y_n \text{ are fresh variables} \\ \sigma &= \{y_1/x_1, \dots, y_n/x_n\} & \sigma' = \{x_1/y_1, \dots, x_n/y_n\} \end{aligned}$$

**Configurations.** The grammar of configurations includes the usual  $\pi$ -calculus forms for composition and name restriction. However, because functional computations return values (which may, in turn, contain channels), we distinguish between the “main” thread  $\bullet M$  (which returns a meaningful value) and the threads  $\circ M$  created by **fork** (which do not).

**Reduction.** Reduction rules for terms and configurations, and equivalences for configurations, are given in Figure 4. Term reduction ( $\longrightarrow_V$ ) is standard call-by-value left-to-right evaluation; the only complication is that application not only substitutes for the  $\lambda$ -bound variables, but also applies any substitution captured by the closure. Configuration equivalence ( $\equiv$ ) is also relatively standard. We provide an additional equivalence observing that **link** is symmetric in order to simplify the definition of reduction. Communication is provided by rule **SEND** and

Term reduction	
$ \begin{aligned} & (\lambda^\sigma x.M) V \longrightarrow_V M(\{V/x\} \uplus \sigma) \\ & (); M \longrightarrow_V M \\ & \text{let } (x, y) = (V, V') \text{ in } M \longrightarrow_V M\{V/x, V'/y\} \\ & \text{case } (\text{inl } V) \{ \text{inl } x \mapsto N; \text{inr } x \mapsto N' \} \longrightarrow_V N\{V/x\} \\ & \quad E[M] \longrightarrow_V E[M'] \quad \text{if } M \longrightarrow_V M' \end{aligned} $	
Configuration equivalence	
$ \begin{aligned} F[\text{link}(x, y)] &\equiv F[\text{link}(y, x)] & C \parallel D &\equiv D \parallel C & C \parallel (D \parallel E) &\equiv (C \parallel D) \parallel E \\ C \parallel (\nu x)D &\equiv (\nu x)(C \parallel D) \text{ if } x \notin \text{fv}(C) & G[C] &\equiv G[D] \text{ if } C \equiv D \end{aligned} $	
Configuration reduction	
$ \text{SEND} \quad \frac{}{F[\text{send}(V, x)] \parallel F'[\text{receive } x] \longrightarrow F[x] \parallel F'[(V, x)]} $	$ \text{LIFT} \quad \frac{C \longrightarrow C'}{G[C] \longrightarrow G[C']} $
$ \text{FORK} \quad \frac{x \text{ is a fresh channel name}}{F[\text{fork}(\lambda^\sigma y.M)] \longrightarrow (\nu x)(F[x] \parallel M(\{x/y\} \uplus \sigma))} $	$ \text{WAIT} \quad \frac{}{(\nu x)(F[\text{wait } x] \parallel \phi x) \longrightarrow F[()]} $
$ \text{LINK} \quad \frac{x \in \text{fv}(M)}{(\nu x)(F[\text{link}(x, y)] \parallel F'[M]) \longrightarrow (\nu x)(F[x] \parallel F'[\text{wait } x; M\{y/x\}])} $	$ \text{LIFTV} \quad \frac{M \longrightarrow_V M'}{G[M] \longrightarrow G[M']} $

**Fig. 4:** Reduction Rules and Equivalences for Terms and Configurations

session initiation is provided by rule FORK. Rule WAIT combines synchronisation of closed channels with garbage collection of the associated name restriction. Rule LINK is complicated by the need to produce a session of type `end`; the inserted `wait` synchronises with the produced session.

**Relation Notation.** We write  $R R'$  for the sequential composition of relations  $R$  and  $R'$ ,  $R \cup R'$  for the union of relations  $R$  and  $R'$ ,  $R^?$  for the reflexive closure of  $R$ ,  $R^+$  for the transitive closure of  $R$ , and  $R^*$  for the reflexive, transitive closure of  $R$ .

**Configuration Typing.** Our syntax of configurations permits various forms of deadlocked configurations. For example, if we define the terms  $M$  and  $N$  by

$$\begin{aligned}
M &\triangleq \text{let } (z, y) = \text{receive } y \text{ in} & N &\triangleq \text{let } (z, x) = \text{receive } x \text{ in} \\
&\quad \text{let } x = \text{send}(z, x) \text{ in } M' & &\quad \text{let } y = \text{send}(z, y) \text{ in } N'
\end{aligned}$$

given suitable terms  $M'$  and  $N'$ , then it is apparent that configurations such as  $(\nu xy)M$ ,  $(\nu xy)(M \parallel M)$  and  $(\nu xy)(M \parallel N)$  cannot reduce further, even though

Configuration typing		
$\frac{\Gamma \vdash M : T \quad T \neq \text{end}_i}{\Gamma \vdash \bullet \bullet M}$	$\frac{\Gamma \vdash M : \text{end}_i}{\Gamma \vdash \circ \circ M}$	$\frac{\Gamma, x : S^\# \vdash^\phi C}{\Gamma \vdash^\phi (\nu x)C}$
$\frac{\Gamma, x : S \vdash^\phi C \quad \Gamma', x : \bar{S} \vdash^{\phi'} C'}{\Gamma, \Gamma', x : S^\# \vdash^{\phi+\phi'} C \parallel C'}$		
Combination of flags		
$\circ + \circ = \circ$	$\circ + \bullet = \bullet$	$\bullet + \circ = \bullet$
$\bullet + \bullet$ undefined		
Reduction of configuration typing environments		
$\frac{}{(!T.S)^\# \longrightarrow S^\#}$	$\frac{}{(?T.S)^\# \longrightarrow S^\#}$	$\frac{T \longrightarrow T'}{\Gamma, x : T \longrightarrow \Gamma, x : T'}$

**Fig. 5:** Configuration Typing

$M$  and  $N$  can be individually well-typed. To exclude such cases, we provide a type discipline for configurations (Figure 5). It is based on type systems for linear  $\pi$ -calculus [14] with two significant differences.

- First, we seek to assure that there is at most one main thread. This constraint is enforced by the flags ( $\bullet$  and  $\circ$ ) on the derivations: a derivation  $\Gamma \vdash \bullet C$  indicates that configuration  $C$  contains the main thread, while  $\Gamma \vdash \circ C$  indicates that  $C$  does not contain the main thread. We write  $\Gamma \vdash C$  to abbreviate  $\exists \phi. \Gamma \vdash^\phi C$ , that is,  $C$  may include a main thread.
- Second, we require that exactly one channel is shared at each composition of processes. This restricts standard type systems for linear  $\pi$ -calculus, which allow an arbitrary number of processes (including none) to be shared at a composition of processes.

We see that the earlier stuck examples are ill-typed in this system:  $(\nu xy)M$  because  $y$  must have a type  $S^\#$  in  $M$ ;  $(\nu xy)(M \parallel M)$  because there is no type  $S^\#$  such that both  $S$  and  $\bar{S}$  are of the form  $?T.S'$ , as required by `receive`; and,  $(\nu xy)(M \parallel N)$  because both  $x$  and  $y$  must be shared between  $M$  and  $N$ , but the typing rule for composition only allows one channel to be shared.

Observe that reduction may transform the types of sessions—for example, if  $\Gamma \vdash F[\text{send}(V, x)] \parallel F'[\text{receive } x]$ , then  $\Gamma$  must contain an assignment  $x : (!T.S)^\#$  for some type  $T$  and session type  $S$ . However, after reduction, for  $F[x] \parallel F'[(V, x)]$  to be well-typed we need  $x : S^\#$ . We capture this constraint with the reduction relation  $\Gamma \longrightarrow \Gamma'$ . Now, we can show that reduction preserves typing. We begin with term reduction.

**Lemma 1.** *If  $\Gamma \vdash M : T$  and  $M \longrightarrow_V M'$ , then  $\Gamma \vdash M' : T$*



The proof is by induction on  $M$ ; the cases are all standard. We can extend this result to show preservation of configuration typing under reduction.

**Theorem 2.** *If  $\Gamma \vdash C$  and  $C \longrightarrow C'$  then there is some  $\Gamma'$  such that  $\Gamma \longrightarrow^? \Gamma'$  and  $\Gamma' \vdash C'$ .*

The proof is in Appendix B.

**Typing and Configuration Equivalence.** Alas, our notion of typing is not preserved by configuration equivalence. For example, assume that  $\Gamma \vdash (\nu xy)(C \parallel (D \parallel E))$ , where  $x \in \text{fv}(C)$ ,  $y \in \text{fv}(D)$ , and  $x, y \in \text{fv}(E)$ . We have that  $C \parallel (D \parallel E) \equiv (C \parallel D) \parallel E$ , but  $\Gamma \not\vdash (\nu xy)((C \parallel D) \parallel E)$ , as both  $x$  and  $y$  must be shared between the processes  $C \parallel D$  and  $E$ . However, we can show that starting from a well-typed configuration, we need never rely on an ill-typed equivalent configuration to expose possible reductions.

**Theorem 3.** *If  $\Gamma \vdash C$ ,  $C \equiv C'$  and  $C' \longrightarrow D'$ , then there exist  $\Gamma', D$  such that  $\Gamma \longrightarrow^? \Gamma'$ ,  $D \equiv D'$ , and  $\Gamma' \vdash D$ .*

*Proof.* Observe that if  $\Gamma \vdash C$ , then for any pair of terms  $M_1, M_2$  appearing in  $C$ , there are environments  $\Gamma_1, \Gamma_2$  and types  $T_1, T_2$  such that  $\Gamma_1 \vdash M_1 : T_1, \Gamma_2 \vdash M_2 : T_2$ , and (because of the typing rule for composition)  $\Gamma_1$  and  $\Gamma_2$  share at most one variable. By examination of the reduction rules, we can conclude that there are well-typed  $C_0, D_0$  such that  $C' = G[C_0]$ ,  $C_0 \longrightarrow D_0$  and  $D' = G[D_0]$ . The result then follows by structural induction on  $C$ , examining the possible equivalences in each case.  $\square$

We extend Theorem 3 to sequences of reductions, defining  $\Longrightarrow$  as  $(\equiv \longrightarrow^* \equiv)^*$ .

**Corollary 4.** *If  $\Gamma \vdash C$  and  $C \Longrightarrow D$ , then there exist  $\Gamma', D'$  such that  $\Gamma \longrightarrow^* \Gamma'$ ,  $D \equiv D'$ , and  $\Gamma' \vdash D'$ .*

### 2.3 Deadlock and its Absence

Previously (§2.2), we saw several examples of deadlocked terms which were rejected by our type system. We now present a general account of deadlock: we characterise deadlocked configurations, and show that well-typed configurations do not evaluate to deadlocked configurations.

We begin by observing that many examples of stuck configurations are already excluded by existing session-typing disciplines: in particular, those configurations in which either too many or too few threads attempt to synchronise on a given session. The cases of interest to us are those in which the threads individually obey the session-typing discipline, but the order of synchronisations in the threads creates deadlock. We say that a thread  $M$  is blocked on a session  $x$ , written  $\text{blocked}(x, M)$ , if  $M$  has evaluated to some context surrounding a communication primitive applied to  $x$ :

$$\begin{aligned} \text{blocked}(x, M) \stackrel{\text{def}}{\iff} \exists N. M = E[\text{send}(N, x)] \\ \vee M = E[\text{receive } x] \\ \vee M = E[\text{wait } x] \end{aligned}$$

In such a case,  $M$  can only reduce further in composition with another thread blocked on  $x$ , and any communication on other sessions in  $M$  will be delayed until a communication on  $x$  has occurred. In general, we abstract over the property that  $y$  depends on  $x$  in  $M$ , abbreviated  $\text{depends}(x, y, M)$ ; in other words,  $M$  is blocked on  $x$ , but has  $y$  as one of its (other) free variables. We extend this notion of dependency from single threads to configurations of threads, with the observation that in a larger configuration intermediate sessions may participate in the dependency.

$$\begin{aligned} \text{depends}(x, y, E[M]) &\stackrel{\text{def}}{\iff} \text{blocked}(x, M) \wedge y \in \text{fv}(E) \\ \text{depends}(x, y, C) &\stackrel{\text{def}}{\iff} (C \equiv G[M] \wedge \text{depends}(x, y, M)) \vee (C \equiv G[D \parallel D'] \\ &\quad \wedge (\exists z. \text{depends}(x, z, D) \wedge \text{depends}(z, y, D'))) \end{aligned}$$

We now define deadlocked configurations as those with cyclic dependencies:

$$\text{deadlocked}(C) \stackrel{\text{def}}{\iff} C \equiv G[D \parallel D'] \wedge \exists x, y. \text{depends}(x, y, D) \wedge \text{depends}(y, x, D').$$

Because the definition of dependency permits intermediate sessions, this definition encompasses cycles involving an arbitrary number of sessions. Finally, we say that a configuration  $C$  is deadlock free if, for all  $D$  such that  $C \implies D$ ,  $\neg \text{deadlocked}(D)$ . Observe that if  $C \equiv D$ , then  $\text{deadlocked}(C) \iff \text{deadlocked}(D)$ .

At this point, we can observe that in any deadlocked configuration, there must be a composition of configurations that shares more than one session. This is precisely the situation that is excluded by our configuration type system.

**Lemma 5.** *If  $\Gamma \vdash C$ , and  $C = G[D \parallel D']$ , then there is exactly one variable  $x$  such that  $\text{fv}(D) \cap \text{fv}(D') = \{x\}$ .*

*Proof.* By structural induction on the derivation of  $\Gamma \vdash C$ ; the only interesting case is for parallel composition, where the desired result is assured by the partitioning of the environment.  $\square$

To extend this observation to deadlock freedom, we must take equivalence into account. While it is true that equivalence need not preserve typing, there are no equivalence rules that affect the free variables of individual threads. Thus, cycles of dependent sessions are preserved by equivalence.

**Lemma 6.** *If  $\Gamma \vdash C$  then  $\neg \text{deadlocked}(C)$ .*

*Proof.* By contradiction. Suppose  $\text{deadlocked}(C)$ , then by expanding the definition of  $\text{deadlocked}$  we know that there must exist variables  $x_1, \dots, x_n$  and processes  $M_1, \dots, M_n$  in  $C$  such that

$$\text{depends}(x_1, x_2, M_1) \wedge \text{depends}(x_2, x_3, M_2) \wedge \dots \wedge \text{depends}(x_n, x_1, M_n).$$

Either  $n = 1$ , which violates linearity, or configuration  $C$  must partition the cycle. However, any cut of the cycle is crossed by at least two channels, so  $C$  must be ill-typed by Lemma 5.  $\square$

Finally, we can combine the previous result with preservation of typing to show that well-typed terms never evaluate to deadlocked configurations.

**Theorem 7.** *If  $\Gamma \vdash M : T$ , then  $\bullet M$  is deadlock-free.*

*Proof.* If  $\Gamma \vdash M : T$ , then  $\Gamma \vdash \bullet M$ , and so  $\neg \text{deadlocked}(\bullet M)$  and, for any  $D$  such that  $\bullet M \Longrightarrow D$ , we know that there is a well-typed  $D' \equiv D$ , and so  $\neg \text{deadlocked}(D)$ .  $\square$

**Progress and Canonical Forms.** We conclude the section by describing a canonical form for configurations, and characterising the stuck terms resulting from the evaluation of well-typed terms. One might hope that evaluation of a well-typed term would always produce a value; however, this is complicated because terms may return sessions. For a simple example, consider the term

$$\bullet \text{fork} (\lambda x. \text{let } (y, x) = \text{receive } x \text{ in send } (y, x))$$

This term spawns a thread (which simply echoes once), and then returns the resulting session; thus, the result of evaluation is a configuration equivalent to

$$(\nu x)(\bullet x \parallel \circ \text{let } (y, x) = \text{receive } x \text{ in send } (y, x)).$$

Clearly, no more evaluation is possible, even though the configuration still contains blocked threads. However, it turns out that we can show that evaluation of terms that do not return sessions must always produce a value.

**Definition 8.** *A process  $C$  is in canonical form if there is some sequence of variables  $x_1, \dots, x_{n-1}$  and terms  $M_1, \dots, M_n$  such that*

$$C = (\nu x_1)(\circ M_1 \parallel (\nu x_2)(\circ M_2 \parallel \dots \parallel (\nu x_{n-1})(\circ M_{n-1} \parallel \phi M_n) \dots)).$$

Note that canonical forms need not be unique. For example, consider the configuration  $\vdash (\nu xy)(C \parallel D \parallel E)$  where  $x \in \text{fv}(C)$ ,  $y \in \text{fv}(D)$ , and  $x, y \in \text{fv}(E)$ . Both  $(\nu x)(C \parallel (\nu y)(D \parallel E))$  and  $(\nu y)(D \parallel (\nu x)(C \parallel E))$  are canonical forms of the original configuration. We can show that any well-typed term must be equivalent to a term in canonical form; again, the key insight is that captured by Lemma 5: if any two sub-configurations share at most one session, then we can order the threads by the sessions they share.

**Lemma 9.** *If  $\Gamma \vdash C$ , then there is some  $C' \equiv C$  such that  $\Gamma \vdash C'$  and  $C'$  is in canonical form.*

The proof is in Appendix B.

We can now state some progress results. We begin with open configurations: each thread must be blocked on either a free variable or a  $\nu$ -bound variable.

**Theorem 10.** *Let  $\Gamma \vdash C$ ,  $C \not\rightarrow$  and let  $C' = (\nu x_1)(\circ M_1 \parallel (\nu x_2)(\circ M_2 \parallel \dots \parallel (\nu x_{n-1})(\circ M_{n-1} \parallel \phi M_n) \dots))$  be a canonical form of  $C$ . Then:*

1. For  $1 \leq i \leq n-1$  either  $\text{blocked}(x_j, M_i)$  where  $j \leq i$  or  $\text{blocked}(y, M_i)$  for some  $y \in \text{dom}(\Gamma)$ ; and,
2. Either  $M_n$  is a value or  $\text{blocked}(y, M_n)$  for some  $y \in \{x_i \mid 1 \leq i \leq n-1\} \cup \text{dom}(\Gamma)$ .

*Proof.* By induction on the derivation of  $\Gamma \vdash C'$ , using the definition of  $\longrightarrow$ .  $\square$

We can strengthen the result significantly when we move to configurations without free variables. To see why, consider just the first two threads of a configuration  $(\nu x_1)(M_1 \parallel (\nu x_2)(M_2 \parallel \dots))$ . As there are no free variables, thread  $M_1$  can only be blocked on  $x_1$ . Now, from the previous result, thread  $M_2$  can be blocked on either  $x_1$  or  $x_2$ . But, were it blocked on  $x_1$ , it could reduce with thread  $M_1$ ; we can conclude it is blocked on  $x_2$ . Generalising this observation gives the following progress result.

**Corollary 11.** *Let  $\vdash C$ ,  $C \not\rightarrow$  and let  $C' = (\nu x_1)(\circ M_1 \parallel (\nu x_2)(\circ M_2 \parallel \dots \parallel (\nu x_{n-1})(\circ M_{n-1} \parallel \phi M_n) \dots))$  be a canonical form of  $C$ . Then:*

1. For  $1 \leq i \leq n-1$ ,  $\text{blocked}(x_i, M_i)$ ; and,
2.  $M_n$  is a value.

*Proof.* By induction on the derivation of  $\vdash P'$ , relying on Theorem 10.  $\square$

Finally, observe that some subset of the variables  $x_1, \dots, x_n$  must appear in the result  $V$ . Therefore, if the original expression returns a value that does not contain any sessions, it will evaluate to a configuration with no blocked threads.

## 2.4 Determinism and Termination

It is straightforward to show that GV is deterministic. In fact, GV enjoys a strong form of determinism, called the diamond property [2].

**Theorem 12.** *If  $\Gamma \vdash C$ ,  $C \equiv \longrightarrow \equiv D_1$ , and  $C \equiv \longrightarrow \equiv D_2$ , then there exists  $D_3$  such that  $D_1 \equiv \longrightarrow \equiv D_3$ , and  $D_2 \equiv \longrightarrow \equiv D_3$ .*

*Proof.* First, observe that  $\longrightarrow_V$  is deterministic, and furthermore configuration reductions always treat  $\longrightarrow_V$  redexes linearly. This means we need only consider the interaction between different configuration reductions. Linear typing ensures that two configuration reductions cannot overlap. Furthermore, each configuration reduction is linear in the existing redexes, so we can straightforwardly perform the reductions in either order.  $\square$

It is not hard to see that the system remains deterministic if we extend the functional part of GV with any well-typed confluent reduction rules at all.

**Theorem 13 (Strong normalisation).** *If  $\Gamma \vdash C$ , then there are no infinite  $\equiv \longrightarrow \equiv$  reduction sequences beginning from  $C$ .*

To prove strong normalisation for core GV, one can use an elementary argument based on linearity. When we add replication (§4.2) and other features, a standard logical relations argument suffices. Strong normalisation also follows as a direct corollary of Theorem 22 and the cut-elimination theorem for classical linear logic.

Syntax			
Types $A, B ::= A \otimes B \mid A \wp B \mid 1 \mid \perp \mid A \oplus B \mid A \& B \mid 0 \mid \top$			
Terms $P, Q ::= x \leftrightarrow y \mid \nu y (P \mid Q) \mid x(y).P \mid x[y].(P \mid Q)$ $\mid x[in_i].P \mid \text{case } x \{P; Q\} \mid x().P \mid x[].0 \mid \text{case } x \{ \}$			
Duality			
$(A \otimes B)^\perp = A^\perp \wp B^\perp$	$1^\perp = \perp$	$(A \oplus B)^\perp = A^\perp \& B^\perp$	$\top^\perp = 0$
$(A \wp B)^\perp = A^\perp \otimes B^\perp$	$\perp^\perp = 1$	$(A \& B)^\perp = A^\perp \oplus B^\perp$	$0^\perp = \top$
Typing			
$\frac{}{x \leftrightarrow w \vdash x : A, w : A^\perp}$	$\frac{P \vdash \Delta, y : A \quad Q \vdash \Delta', y : A^\perp}{\nu y (P \mid Q) \vdash \Delta, \Delta'}$	$\frac{}{x[].0 \vdash x : 1}$	
$\frac{P \vdash \Delta, y : A, x : B}{x(y).P \vdash \Delta, x : A \wp B}$	$\frac{P \vdash \Delta, y : A \quad Q \vdash \Delta', x : B}{x[y].(P \mid Q) \vdash \Delta, \Delta', x : A \otimes B}$	$\frac{P \vdash \Delta}{x().P \vdash \Delta, x : \perp}$	
$\frac{P \vdash \Delta, x : A_i}{x[in_i].P \vdash \Delta, x : A_1 \oplus A_2}$	$\frac{P \vdash \Delta, x : A \quad Q \vdash \Delta, x : B}{\text{case } x \{P; Q\} \vdash \Delta, x : A \& B}$	$\frac{}{\text{case } x \{ \} \vdash \Delta, x : \top}$	

**Fig. 6:** CP Syntax and Typing

### 3 Classical Linear Logic

Figure 6 gives the syntax and typing rules for the multiplicative-additive fragment of CP; we let  $\Delta$  range over typing environments. CP types and duality are the standard propositions and duality function of classical linear logic, while the terms are based on a subset of the  $\pi$ -calculus. The types  $\&$  and  $\oplus$  are interpreted as external and internal choice; the types  $\wp$  and  $\otimes$  are interpreted as input and output, while their units  $\perp$  and  $1$  are interpreted as nullary input and output. Note that CP's typing rules implicitly rebind identifiers: for example, in the hypothesis of the rule for  $\wp$ ,  $x$  identifies a proof of  $B$ , while in the conclusion it identifies a proof of  $A \wp B$ .

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 captures syntactically 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  $P$ ). We will frequently write  $x\langle y \rangle.P$  to abbreviate  $x[z].(y \leftrightarrow z \mid P)$  (see Appendix A for more discussion of output in CP).

Structural congruence

$$\begin{aligned}
x \leftrightarrow w &\equiv w \leftrightarrow x \\
\nu y (P \mid Q) &\equiv \nu y (Q \mid P) \\
\nu y (P \mid \nu z (Q \mid R)) &\equiv \nu z (\nu y (P \mid Q) \mid R), \quad \text{if } y \notin \text{fv}(R) \\
\nu x (P_1 \mid Q) &\equiv \nu x (P_2 \mid Q), \quad \text{if } P_1 \equiv P_2
\end{aligned}$$

Primary cut reduction rules

$$\begin{aligned}
\nu x (w \leftrightarrow x \mid P) &\longrightarrow_C P[w/x] \\
\nu x (x[y].(P \mid Q) \mid x(y).R) &\longrightarrow_C \nu x (Q \mid \nu y (P \mid R)) \\
\nu x (x[], 0 \mid x().P) \vdash \Delta &\longrightarrow_C P \\
\nu x (x[in_i].P \mid \text{case } x \{Q_1; Q_2\}) &\longrightarrow_C \nu x (P \mid Q_i) \\
\nu x (P_1 \mid Q) &\longrightarrow_C \nu x (P_2 \mid Q), \quad \text{if } P_1 \longrightarrow_C P_2
\end{aligned}$$

Commuting conversions

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

**Fig. 7:** CP Congruences and Cut Reduction

**Cut Elimination.** The semantics of CP terms are given by cut reduction, as shown in Figure 7. We write  $\text{fv}(P)$  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  $\longrightarrow_C$  for the cut reduction relation,  $\longrightarrow_{CC}$  for the commuting conversion relation, and  $\longrightarrow$  for  $\longrightarrow_C \cup \longrightarrow_{CC}$ . 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  $\wp$  against  $\otimes$  is more complex than synchronisation of input and output, as it must also manipulate the implicit name restriction and composition in CP's output term. The commuting conversions do not correspond to any reduction rule in  $\pi$ -calculus. We write  $\Longrightarrow$  for  $(\equiv \longrightarrow \equiv)^+$ ,  $\Longrightarrow_C$  for  $(\equiv \longrightarrow_C \equiv)^+$ , and  $\Longrightarrow_{CC}$  for  $\Longrightarrow_C (\longrightarrow_{CC}^*)$ .

Just as cut elimination in logic shows that any proof can be transformed to one that does not make use cut, the reduction rules of CP transform any term into a term blocked on an external communication—that is to say, if  $P \vdash \Delta$ , then  $P \Longrightarrow_{CC} P'$  where  $P' \neq \nu x (Q \mid Q')$  for any  $x, Q, Q'$ . The final commuting conversions play a central role in this transformation, moving any further internal communication behind the external communication.

On types

$$\begin{aligned} \langle A \otimes B \rangle &= !\langle A \rangle. \langle B \rangle & \langle 1 \rangle &= \text{end}_! & \langle A \oplus B \rangle &= \langle A \rangle \oplus \langle B \rangle & \langle 0 \rangle &= \oplus \{ \} \\ \langle A \wp B \rangle &= ?\langle A \rangle. \langle B \rangle & \langle \perp \rangle &= \text{end}_? & \langle A \& B \rangle &= \langle A \rangle \& \langle B \rangle & \langle \top \rangle &= \& \{ \} \end{aligned}$$

On terms

$$\begin{aligned} \langle \nu x (P \mid Q) \rangle &= \text{let } x = \text{fork } (\lambda x. \langle P \rangle) \text{ in } \langle Q \rangle \\ \langle x \leftrightarrow y \rangle &= \text{link } (x, y) \\ \langle x[y].(P \mid Q) \rangle &= \text{let } x = \text{send } (\text{fork } (\lambda y. \langle P \rangle), x) \text{ in } \langle Q \rangle \\ \langle x(y).P \rangle &= \text{let } (y, x) = \text{receive } x \text{ in } \langle P \rangle \\ \langle x[] . 0 \rangle &= x \\ \langle x().P \rangle &= \text{let } () = \text{wait } x \text{ in } \langle P \rangle \\ \langle x[l].P \rangle &= \text{let } x = \text{select } l \text{ } x \text{ in } \langle P \rangle \\ \langle \text{case } x \{ P; Q \} \rangle &= \text{offer } x \{ \text{inl } x \mapsto \langle P \rangle; \text{inr } x \mapsto \langle Q \rangle \} \\ \langle \text{case } x \{ \} \rangle &= \text{let } (y, x) = \text{receive } x \text{ in absurd } y \\ \mathcal{C} \langle \nu x (P \mid Q) \rangle &= (\nu x) (\mathcal{C} \langle P \rangle \parallel \mathcal{C} \langle Q \rangle) \\ \mathcal{C} \langle P \rangle &= \circ \langle P \rangle, \quad P \text{ is not a cut} \end{aligned}$$

**Fig. 8:** Translation of CP Terms into GV

### 3.1 Translation from CP to GV

In this section, we show that GV can simulate CP. Figure 8 gives the translation of CP into GV; typing environments are translated by the pointwise extension of the translation on types. We rely on our encoding of choice in GV (§2.1).

In translating CP terms to GV terms, the key observation is that CP terms contain their continuations; for example, the translation of input includes both a call to `receive` and the translation of the continuation. Additionally, the rebinding that is implicit in CP syntax is made explicit in GV. The translation  $\mathcal{C}(\_)$  translates top-level composition (i.e., composition not under any prefix) to GV configurations; cuts that appear under prefixes are translated to applications of `fork`. As CP processes do not have return values, the translation of a CP process contains no main thread.

It is straightforward to see that the translation preserves typing; note that the channels in the CP typing environment become free variables in its translation.

**Theorem 14.** *If  $P \vdash \Delta$  then  $\langle \Delta \rangle \vdash^\circ \mathcal{C} \langle P \rangle$ .*

Structural congruence in CP is a subset of the structural congruence relation for GV configurations; thus the translation trivially preserves congruence.

**Theorem 15.** *If  $P \equiv Q$ , then  $\mathcal{C} \langle P \rangle \equiv \mathcal{C} \langle Q \rangle$ .*

Finally, observe that the translation of any prefixed CP term is a GV thread of either the form  $F[K \ M]$  for  $K \in \{\text{send}, \text{receive}, \text{wait}\}$  or is  $\circ x$  for some variable  $x$ . Thus, we can see that any cut reduction immediately possible for a process  $P$  is similarly possible for  $\langle P \rangle$ . Following such a reduction, several additional GV reductions may be necessary to expose the next possible communication,

Session types			
$\llbracket !T.S \rrbracket = \llbracket T \rrbracket^\perp \otimes \llbracket S \rrbracket$	$\llbracket ?T.S \rrbracket = \llbracket T \rrbracket \wp \llbracket S \rrbracket$	$\llbracket \text{end}_! \rrbracket = 1$	$\llbracket \text{end}_? \rrbracket = \perp$
Functional types			
$\llbracket T \rrbracket = \llbracket T \rrbracket^\perp$ , if $T$ is not a session type			
$\llbracket \mathbf{0} \rrbracket = \llbracket 0 \rrbracket$	$\llbracket \mathbf{1} \rrbracket = \llbracket 1 \rrbracket$		
$\llbracket T + U \rrbracket = \llbracket T \rrbracket \oplus \llbracket U \rrbracket$	$\llbracket T \times U \rrbracket = \llbracket T \rrbracket \otimes \llbracket U \rrbracket$		
$\llbracket T \multimap U \rrbracket = \llbracket T \rrbracket^\perp \wp \llbracket U \rrbracket$			
$\llbracket S \rrbracket = \llbracket S \rrbracket$			

**Fig. 9:** Translation of GV Types into CP

such as substituting the received values into the continuation in the case of the translation of input, or spawning new threads in the translation of composition.

**Theorem 16.** *If  $P \vdash \Delta$  and  $P \longrightarrow_C Q$ , then  $\mathcal{C}(P) \longrightarrow^+ \mathcal{C}(Q)$ .*

*Proof.* By induction on  $P$ ; the cases are all straightforward.

The commuting conversions in CP do not expose additional reductions, but are only necessary to assure that the result of evaluation does not have a cut at the top level. Our characterisation of deadlock freedom in GV has no such requirement, so we have no need of corresponding steps in GV.

### 3.2 Translation from GV to CP

In this section, we show that CP can simulate GV. Figure 9 gives the translation on types and Figure 10 gives the translation on terms, substitutions, and configurations; we translate typing environments by the pointwise extension of the translation on types.

The translation on session types is homomorphic except for output, where the output type is dualised. This accounts for the discrepancy between  $\overline{!T}.S = ?T.S$  and  $(A \otimes B)^\perp = A^\perp \wp B^\perp$ . Following our previous work [16], the translation on functional types is factored through an auxiliary translation  $\llbracket - \rrbracket$ . The intuition is that the translation  $\llbracket T \rrbracket$  of a functional type  $T$  is the type of its *interface*, whereas  $\llbracket T \rrbracket^\perp$  is the type of its *implementation*.

As CP processes do not have return values, the translation  $\llbracket M \rrbracket z$  of a term  $M$  of type  $T$  includes the additional argument  $z : \llbracket T \rrbracket^\perp$ , which is a channel for simulating the return value. The translation on session terms is somewhat complicated by the need to include apparently trivial axiom cuts (highlighted in grey). These are needed to align with the translation of values, which permit further reduction inside the value constructors. The output in the translation of a fork arises from the need to apply the argument to a freshly generated channel (notice that application is simulated by an output). Linking is simulated by a



#### Session terms

$$\begin{aligned}
\llbracket \text{fork } M \rrbracket z &= \nu w (w \leftrightarrow z \mid \nu x (\llbracket M \rrbracket x \mid \nu y (x \langle w \rangle . x \leftrightarrow y \mid y[]))) \\
\llbracket \text{link } (M, N) \rrbracket z &= \nu v (v \leftrightarrow z \mid \nu w (v \leftrightarrow w \mid \nu x (\llbracket M \rrbracket x \mid \nu y (\llbracket N \rrbracket y \mid w().x \leftrightarrow y)))) \\
\llbracket \text{send } (M, N) \rrbracket z &= \nu x (\llbracket N \rrbracket x \mid \nu y (\llbracket M \rrbracket y \mid x \langle y \rangle . x \leftrightarrow z)) \\
\llbracket \text{receive } M \rrbracket z &= \nu y (\llbracket M \rrbracket y \mid y(x). \nu w (w \leftrightarrow y \mid z \langle x \rangle . w \leftrightarrow z)) \\
\llbracket \text{wait } M \rrbracket z &= \nu y (y \leftrightarrow z \mid \llbracket M \rrbracket y)
\end{aligned}$$

#### Functional terms

$$\begin{aligned}
\llbracket x \rrbracket z &= x \leftrightarrow z \\
\llbracket \lambda^\sigma x. M \rrbracket z &= \llbracket \sigma \rrbracket (z(x). \llbracket M \rrbracket z) \\
\llbracket L M \rrbracket z &= \nu x (\llbracket M \rrbracket x \mid \nu y (\llbracket L \rrbracket y \mid y \langle x \rangle . y \leftrightarrow z)) \\
\llbracket () \rrbracket z &= z[] \\
\llbracket \text{let } () = M \text{ in } N \rrbracket z &= \nu y (\llbracket M \rrbracket y \mid y(). \llbracket N \rrbracket z) \\
\llbracket (M, N) \rrbracket z &= \nu x (\llbracket M \rrbracket x \mid \nu y (\llbracket N \rrbracket y \mid z \langle x \rangle . y \leftrightarrow z)) \\
\llbracket \text{let } (x, y) = M \text{ in } N \rrbracket z &= \nu y (\llbracket M \rrbracket y \mid y(x). \llbracket N \rrbracket z) \\
\llbracket \text{inl } M \rrbracket z &= \nu x (\llbracket M \rrbracket x \mid z[in_1].x \leftrightarrow z) \\
\llbracket \text{inr } M \rrbracket z &= \nu x (\llbracket M \rrbracket x \mid z[in_2].x \leftrightarrow z) \\
\llbracket \text{case } L \{ \text{inl } x \mapsto M; \text{inr } x \mapsto N \} \rrbracket z &= \nu x (\llbracket L \rrbracket x \mid \text{case } x \{ \llbracket M \rrbracket z; \llbracket N \rrbracket z \}) \\
\llbracket \text{absurd } L \rrbracket z &= \nu x (\llbracket L \rrbracket x \mid \text{case } x \{ \})
\end{aligned}$$

#### Substitutions

$$\begin{aligned}
\llbracket \{ V_i / x_i \} \rrbracket (P) &= \hat{\nu} (x_i \mapsto \llbracket V_i \rrbracket x_i)_i [P] \\
\hat{\nu} (x_i \mapsto P_i)_i [P] &\triangleq \nu x_1 (P_1 \mid \dots \nu x_n (P_n \mid P) \dots)
\end{aligned}$$

#### Configurations

$$\begin{aligned}
\llbracket \circ M \rrbracket z &= \nu y (\llbracket M \rrbracket y \mid y[]) \\
\llbracket \bullet M \rrbracket z &= \llbracket M \rrbracket z \\
\llbracket (\nu x) C \rrbracket z &= \llbracket C \rrbracket z \\
\llbracket C \parallel_x C' \rrbracket z &= \nu x (\llbracket C \rrbracket z \mid \llbracket C' \rrbracket z)
\end{aligned}$$

**Fig. 10:** Translation of GV Terms, Substitutions, and Configurations into CP

link ( $\leftrightarrow$ ) guarded by a nullary input which matches the nullary output of the output channel. Sending is simulated by output as one might expect. Receiving is simulated by input composed with sending the result to the return channel. Waiting is simulated by simply connecting the result to the return channel.

Variables are linked to the return channel. Closures are simulated by input, subject to an appropriate substitution, and application by output. Unit values are simulated by empty output to the return channel. Pairs are simulated by evaluating both components in parallel and transmitting the results to the return channel. Injections are simulated by injections. Each elimination form (other than application) guards the continuation with a suitable prefix, delaying reduction of the continuation until a value has been computed to pass to it.

The translation of configurations is quite direct. We write  $C \parallel_x C'$  to indicate that the variable  $x$  is shared by  $C$  and  $C'$ ; in a well-typed GV configuration, there will always be exactly one such variable, so the translation is unambiguous.

Our translation differs from both Wadler's [22] and our previous one [16], neither of which simulate even plain  $\beta$ -reduction. This is because the obvious translation to CP cannot simulate substitution under a lambda abstraction, motivating our use of closures / weak explicit substitution. Indeed, others have taken advantage of full explicit substitutions in order to simulate small-step semantics of  $\lambda$ -calculi in the full  $\pi$ -calculus [20].

Another departure from the previous translations to CP is that, despite the call-by-value semantics of GV, our translation is more in the spirit of call-by-name. For instance, in the translation of an application  $LM$ , the evaluation of  $L$  and  $M$  can happen in parallel, and  $\beta$ -reduction can occur before  $M$  has reduced to a value. The previous translations hide the evaluation of  $M$  behind the prefix  $y\langle x \rangle$ , which means that reduction of  $M$  can get stuck in the case that  $L$  is a free variable. Short of performing CPS transformation on the translation, our new approach seems necessary in order to ensure that  $\llbracket - \rrbracket$  preserves reduction.

It is straightforward to show that the translation preserves typing.

**Theorem 17.**

1. If  $\Gamma \vdash M : T$ , then  $\llbracket M \rrbracket z \vdash \llbracket \Gamma \rrbracket, z : \llbracket T \rrbracket^\perp$ .
2. If  $\Gamma \vdash C$ , then  $\exists T. \llbracket C \rrbracket z \vdash \llbracket \Gamma \rrbracket, z : \llbracket T \rrbracket^\perp$ .

*Proof.* By induction on derivations. □

We now show that reduction in GV is preserved by reduction in CP. First, we observe that structural equivalence is preserved.

**Theorem 18.** If  $\Gamma \vdash C$ ,  $\Gamma \vdash D$ , and  $C \equiv D$ , then  $\llbracket C \rrbracket z \equiv \llbracket D \rrbracket z$ .

*Proof.* By induction on the derivation of  $\Gamma \vdash C$ . □

As the translation on terms and configurations are compositional, we can mechanically lift them to translations on evaluation contexts and configuration contexts such that the following lemma holds by construction. Each translation of a context takes two arguments: a function that describes the CP term to plug into the hole, and an output channel.

**Lemma 19.** For  $X \in \{E, F, G\}$ ,  $\llbracket X[M] \rrbracket z = \llbracket X \rrbracket \llbracket M \rrbracket z$

We will make implicit use of Lemma 19 throughout our proofs. We write  $x \mapsto P$  for a function that maps a name  $x$  to a process  $P$  that depends on  $x$ .

We now show that substitution commutes with  $\llbracket - \rrbracket$ .

**Lemma 20.** If  $\Gamma \vdash M : T$ ,  $\Gamma \vdash \sigma : \Delta$ , and  $z \notin \text{dom}(\sigma)$ , then  $\llbracket \sigma \rrbracket (\llbracket M \rrbracket z) \implies \llbracket M\sigma \rrbracket z$ .

*Proof.* By induction on the structure of  $M$ . Here we show the cases for variables and closures.

- Case  $x$ . By linearity there exists  $V$  such that  $\sigma = \{V/x\}$ .

$$\llbracket \sigma \rrbracket (\llbracket x \rrbracket z) = \nu x (\llbracket V \rrbracket x \mid x \leftrightarrow z) \longrightarrow \llbracket V \rrbracket z = \llbracket x\sigma \rrbracket z$$

– Case  $\lambda^{\sigma'} x.M$ .

$$\begin{aligned}
& \llbracket \sigma \rrbracket (\llbracket \lambda^{\sigma'} x.M \rrbracket) \\
&= (\sigma' = \{V_i/x_i\}_i) \\
& \llbracket \sigma \rrbracket (\hat{\nu}(x_i \mapsto (\llbracket V_i \rrbracket x_i))_i [z(x). \llbracket M \rrbracket z]) \\
&= (\sigma = \sigma_1 \uplus \dots \uplus \sigma_n \text{ where } \text{dom}(\sigma_i) = \text{fv}(V_i)) \\
& \llbracket \sigma_1 \rrbracket (\dots \llbracket \sigma_n \rrbracket (\hat{\nu}(x_i \mapsto \llbracket V_i \rrbracket x_i)_i [z(x). \llbracket M \rrbracket z]))) \\
&= (\text{structural equivalence}) \\
& \hat{\nu}(x_i \mapsto \llbracket \sigma_i \rrbracket (\llbracket V_i \rrbracket x_i))_i [z(x). \llbracket M \rrbracket z] \\
&\implies (\text{IH}) \\
& \hat{\nu}(x_i \mapsto \llbracket V_i \sigma_i \rrbracket x_i)_i [z(x). \llbracket M \rrbracket z] \\
&= (V_i \sigma_i = V \sigma) \\
& \hat{\nu}(x_i \mapsto \llbracket V_i \sigma \rrbracket x_i)_i [z(x). \llbracket M \rrbracket z] \\
&= (\text{definition of } \llbracket - \rrbracket) \\
& \llbracket \lambda^{\sigma'} x.M \rrbracket \\
&= (\text{definition of substitution}) \\
& \llbracket \lambda^{\sigma'} x.M \sigma \rrbracket
\end{aligned}$$

Each of the remaining non-binding form cases follows straightforwardly using the induction hypothesis. Each of the remaining binding form cases requires a commuting conversion to push the appropriate substitution through a prefix.  $\square$

Using the substitution lemma, we prove that  $\llbracket - \rrbracket$  preserves reduction on terms.

**Theorem 21.** *If  $\Gamma \vdash M$ ,  $\Gamma \vdash N$ , and  $M \longrightarrow_V N$ , then  $\llbracket M \rrbracket z \implies \llbracket N \rrbracket v$ .*

*Proof.* By induction on the derivation of  $M \longrightarrow_V N$ . Here we show the case of  $\beta$ -reduction.

– Case  $(\lambda^\sigma x.M) V \longrightarrow_V M(\{V/x\} \cup \sigma)$ .

$$\begin{aligned}
& \llbracket (\lambda^\sigma x.M) V \rrbracket z \\
&= (\text{definition of } \llbracket - \rrbracket) \\
& \nu w (\llbracket V \rrbracket w \mid \nu y (\llbracket \sigma \rrbracket (y(x). \llbracket M \rrbracket y) \mid y[x](w \leftrightarrow x \mid y \leftrightarrow z))) \\
&\implies_C (\text{cut send against receive}) \\
& \nu w (\llbracket V \rrbracket w \mid \nu y (y \leftrightarrow z \mid \nu x (w \leftrightarrow x \mid \llbracket \sigma \rrbracket (\llbracket M \rrbracket y)))) \\
&\implies_C (\text{cut links and } \alpha \text{ rename}) \\
& \nu x (\llbracket V \rrbracket x \mid \llbracket \sigma \rrbracket (\llbracket M \rrbracket z)) \\
&\implies (\text{by Lemma 20}) \\
& \llbracket M(\{V/x\} \uplus \sigma) \rrbracket
\end{aligned}$$

The remaining base cases are similarly direct. The inductive case for reduction inside an evaluation context follows straightforwardly by observing that the translation of an evaluation context never places its argument inside a prefix.  $\square$

Finally, we prove that  $\llbracket - \rrbracket$  preserves reduction on configurations.

**Theorem 22.** *If  $\Gamma \vdash C$ ,  $\Gamma' \vdash D$ ,  $\Gamma \longrightarrow \Gamma'$ , and  $C \longrightarrow D$ , then  $\llbracket C \rrbracket z \implies \llbracket D \rrbracket z$ .*

Syntax	Session types $S ::= !T.S \mid ?T.S \mid \text{end} \mid S^\sharp$
	Constants $K ::= \text{send} \mid \text{receive} \mid \text{fork} \mid \text{close} \mid \text{link}$
Changes to duality	$\overline{\text{end}} = \text{end}$
Changes to type schemas for constants	
	$\text{fork} : (S \multimap \mathbf{1}) \multimap \overline{S} \quad \text{close} : \text{end} \multimap \mathbf{1} \quad \text{link} : S \times \overline{S} \multimap \mathbf{1}$

**Fig. 11:** Syntax and Typing Rules for Combined Closed Channels

*Proof.* By induction on the derivation of  $C \longrightarrow D$ . The inductive cases follow straightforwardly from the compositionality of the definitions and Theorem 21. We give a representative example of the base cases. The other cases appear in Appendix B.

- Case  $F[\text{send}(V, x)] \parallel F'[\text{receive } x] \longrightarrow F[x] \parallel F'[(V, x)]$ .

$$\begin{aligned}
& \llbracket F[\text{send}(V, x)] \parallel F'[\text{receive } x] \rrbracket z \\
&= \\
& \nu x (\llbracket F \rrbracket [\llbracket \text{send}(V, x) \rrbracket y] z \mid \llbracket F' \rrbracket [\llbracket \text{receive } x \rrbracket] z) \\
&= \\
& \nu x (\llbracket F \rrbracket [z \mapsto \nu v (\llbracket V \rrbracket v \mid \nu w (x \leftrightarrow w \mid w \langle v \rangle . w \leftrightarrow z))] z \\
& \quad \mid \llbracket F' \rrbracket [z \mapsto \nu y (x \leftrightarrow y \mid y \langle v \rangle . \nu w (y \leftrightarrow w \mid z \langle v \rangle . w \leftrightarrow z))] z) \\
&\Longrightarrow_C (\text{cut links}) \\
& \nu x (\llbracket F \rrbracket [z \mapsto \nu v (\llbracket V \rrbracket v \mid x \langle v \rangle . x \leftrightarrow z)] z \\
& \quad \mid \llbracket F' \rrbracket [z \mapsto x \langle v \rangle . \nu w (x \leftrightarrow w \mid z \langle v \rangle . w \leftrightarrow z)] z) \\
&\Longrightarrow_C (\text{cut send against receive}) \\
& \nu x (\llbracket F \rrbracket [z \mapsto \nu v (\llbracket V \rrbracket v \mid x \leftrightarrow z)] \mid \llbracket F' \rrbracket [z \mapsto \nu w (x \leftrightarrow w \mid z \langle v \rangle . w \leftrightarrow z)] z) \\
&\equiv \\
& \nu x (\llbracket F \rrbracket [z \mapsto x \leftrightarrow z] z \mid \llbracket F' \rrbracket [z \mapsto \nu v (\llbracket V \rrbracket v \mid \nu w (x \leftrightarrow w \mid z \langle v \rangle . w \leftrightarrow z))] z) \\
&= \\
& \llbracket F[x] \parallel F'[(V, x)] \rrbracket
\end{aligned}$$

□

## 4 Extending GV

In this section, we consider two extensions of our core calculus: in the first, we have a single, self-dual type for closed sessions; in the second, we support unlimited types. In both cases, we are able to directly apply our characterisation of deadlock to show that the extended systems are deadlock free.

### 4.1 Unifying $\text{end}_?$ and $\text{end}_!$

We begin by defining a language, based on GV, but combining the types  $\text{end}_?$  and  $\text{end}_!$  of closed sessions. Figure 11 gives the alterations to the syntax and typing

Extended configuration equivalence
$C \parallel \circ () \equiv C$
Extended reduction rules (all other reduction rules apply as in GV)
<div style="text-align: center;"> CLOSE  <math display="block">\frac{}{(\nu x)(F[\text{close } x] \parallel F'[\text{close } x]) \longrightarrow F[()] \parallel F'[()]}</math> </div>
<div style="text-align: center;"> LINK  <math display="block">\frac{x \in \text{fv}(P)}{F[\text{link}(x, y)] \parallel C \longrightarrow F[()] \parallel C\{y/x\}}</math> </div>

**Fig. 12:** Updated Configuration Evaluation Rules

rules. The session types are updated to have a single, self-dual type `end`; a new constant, `close` is provided to eliminate sessions of type `end`. (In many existing systems, sessions of type `end` are treated as unlimited, subject to weakening, rather than providing an explicit `close`. We have left `close` explicit to simplify the presentation.) The type schemas for `fork` and `link` have been simplified, as we no longer need to build the elimination of closed sessions into `fork`. Figure 12 gives the updated evaluation rules for the extended language. In addition to a new rule for `close` (replacing the rule for `wait`), the rule for `link` can be simplified significantly (as it can now return a unit value instead of a closed channel).

Our extended language is, perhaps surprisingly, strictly more expressive than GV. To see this, we can consider the following term (assuming well-typed  $M, N$ ):

$$\text{let } w = \text{fork}(\lambda w. \text{close } w; M) \text{ in close } w; N$$

Initially, the forked thread and its parent share (only) session  $w$ . After both threads close  $w$ , however, there can be no further communication between the threads; in contrast, in core GV, there is always a final synchronisation with `wait`. To capture the increase in expressivity, we must extend the existing configuration typing rules (Figure 5) with a rule for composition in which no threads are shared (such as after the reduction of `close`  $w$  in the example):

$$\frac{\Gamma \vdash^\phi C \quad \Gamma' \vdash^{\phi'} C'}{\Gamma, \Gamma' \vdash^{\phi+\phi'} C \parallel C'}$$

We have a slightly different result about channels shared between processes, to account for the multiple possible typings of composition:

**Lemma 23.** *If  $\Gamma \vdash C$  and  $C = G[D \parallel D']$ , then  $\text{fv}(D) \cap \text{fv}(D')$  is either empty or the singleton set  $\{x\}$  for some variable  $x$ .*

The proof is again by induction on the derivation of  $\Gamma \vdash C$ . Clearly, this change does not allow the introduction of cyclic dependencies.

**Theorem 24.** *If  $\Gamma \vdash M : T$  then  $\bullet M$  is deadlock free.*

Syntactic extensions	
Types	$T ::= \Box T \mid \dots$
Terms	$M, N ::= \text{let } !x = M \text{ in } N \mid !M \mid \dots$
Values	$V ::= !V \mid \dots$
Evaluation contexts	$E ::= \text{let } !x = E \text{ in } M \mid !E \mid \dots$
Typing rules	
$\frac{\Gamma \vdash M : T \quad \Box \Gamma}{\Gamma \vdash !M : \Box T}$	$\frac{\Gamma \vdash M : \Box T \quad \Gamma', x : T \vdash N : U}{\Gamma \vdash \text{let } !x = M \text{ in } N : U}$
$\frac{\Gamma \vdash M : T}{\Gamma, x : \Box U \vdash M : T}$	$\frac{\Gamma, x : \Box T, x' : \Box T \vdash M : U}{\Gamma, x : \Box T \vdash M\{x/x'\} : T}$
Reduction	
$\text{let } !x = !V \text{ in } M \longrightarrow_V M\{V/x\}$	

**Fig. 13:** GV Extensions for Unlimited Types

## 4.2 Adding Unlimited Types

So far, we have limited GV to containing only linear types. In this section, we consider one standard approach to extending the term language to include unlimited types. We will then show we can encode replicated processes using term-level contraction and weakening. Finally, we will show that our encoding is equivalent to the replicated processes in CP.

Figure 13 gives the extension of GV. We begin by adding a new class of types,  $\Box T$ , representing unlimited types. (The typical notation for such types in linear logic,  $!T$ , clashes with the notation for output in session types.) We will write  $T \rightarrow U$  to abbreviate  $\Box(T \multimap U)$ . We add terms to construct and deconstruct values of type  $\Box T$ ;  $\Box \Gamma$  denotes that every type in  $\Gamma$  must be of the form  $\Box U$  for some type  $U$ . Finally, we allow values of type  $\Box T$  to be weakened (discarded) and contracted (duplicated). The reduction rule for  $\Box T$  values is unsurprising—however, unlike in the other reductions,  $x$  be used non-linearly in  $M$ . The extension of deadlock freedom to this calculus is immediate.

Wadler’s CP calculus provides unlimited service channels, which can be used to obtain arbitrarily many copies of some concurrent behavior. We can encode such channels as follows. First, we introduce new, dual session type constructors  $Service(S)$  and  $Server(S)$ , defined by

$$Server(S) \triangleq !(\overline{S} \rightarrow \text{end}_!).\text{end}_! \quad Service(S) \triangleq ?(S \rightarrow \text{end}_!).\text{end}_!$$

Note that  $\overline{Server(S)} = Service(\overline{S})$ . Note the parallel between the encoding of session-level sums using value-level sums and the encoding of session-level replication using value-level replication—in each case, the key is “trivial” channels  $!T.\text{end}_!$  for suitable  $T$ . We then introduce new constants `replicate` and `request`,

with the types

$$\text{replicate} : ((S \rightarrow \text{end}_!) \times \text{Server}(S)) \multimap \text{end}_! \quad \text{request} : \text{Service}(S) \multimap S$$

defined as follows:

$$\begin{aligned} \text{replicate}(f, s) &\triangleq \text{link}(s, \text{fork}(\lambda x. \text{send}(f, x))) \\ \text{request } s &\triangleq \text{let } (f, s) = \text{receive } s \text{ in wait } s; \text{let } !g = f \text{ in } g \end{aligned}$$

Now, we can extend the translation from GV into CP:

$$\begin{aligned} \llbracket !M \rrbracket z &\triangleq \nu x (!z(y).y \leftrightarrow x \mid \llbracket M \rrbracket x) \\ \llbracket \text{let } !x = V \text{ in } M \rrbracket z &\triangleq \nu y (\llbracket V \rrbracket y \mid ?y[x].\llbracket M \rrbracket z) \end{aligned}$$

and similarly the translation from CP into GV:

$$\begin{aligned} \llbracket !s(x).P \rrbracket &\triangleq \text{replicate}(\lambda x. \llbracket P \rrbracket, s) \\ \llbracket ?s[x].P \rrbracket &\triangleq \text{let } x = \text{request } s \text{ in } \llbracket P \rrbracket \\ \left( \frac{P \vdash \Delta}{P \vdash \Delta, x : ?A} \right) &\triangleq \text{let } (y, s) = \text{receive } s \text{ in} \\ &\quad \text{wait } s; \llbracket P \rrbracket \\ \left( \frac{P \vdash \Delta, x : ?A, y : ?A}{P[x/y] \vdash \Delta, x : ?A} \right) &\triangleq \text{let } f = \text{request } x \text{ in} \\ &\quad \text{let } x = \text{fork}(\lambda x. \text{replicate}(f, x)) \text{ in} \\ &\quad \text{let } y = \text{fork}(\lambda x. \text{replicate}(f, y)) \text{ in } \llbracket P \rrbracket \end{aligned}$$

The translation from CP into GV emphasises that, while weakening and contraction are implicit in CP syntax, they play a central role in the CP semantics of exponentials, and thus have non-trivial translation to GV. Finally, the extensions of Theorems 16 and 22 to unlimited channels is direct.

## 5 Related Work

**Session Types and Functional Languages.** Session types were originally proposed by Honda [10], and later extended by Takeuchi et al. [18] and by Honda et al. [11]. Honda's system relies on a substructural type system (in which sessions cannot be duplicated or discarded) and adopts the syntax  $\&$  and  $\oplus$  for choice; however, he does not draw a connection between his type system and the connectives of linear logic, and his system includes a single, self-dual closed channel. Vasconcelos et al. [21] develop a language that integrates session-typed communication primitives and a functional language. Gay and Vasconcelos [8] extend the approach to describe asynchronous communication with statically-bounded buffers. Their approach provides a more flexible mechanism of session initiation, distinct from their construct for thread creation, and they do not consider deadlock. Kobayashi [12] describes an embedding of session-typed  $\pi$ -calculus in polyadic linear  $\pi$ -calculus, relying on multi-argument send and receive to capture the state of a communication and variant types to capture choice; Dardha et al. [7] extend his approach to subtyping and polymorphism.

**Linear Logic and Session Types.** When he originally described linear logic, Girard [9] suggested that it would be suited to reasoning about concurrency. Abramsky [1] and Bellin and Scott [3] 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.

Caires and Pfenning [4] present a session type system for  $\pi$ -calculus that exactly corresponds to the proof system for the dual intuitionistic linear logic, and show that (up to congruence) cut reductions corresponds to process reductions or process equivalences. Toninho et al. [19] consider embeddings of the  $\lambda$ -calculus into session-typed  $\pi$ -calculus; their focus is on expressing the concurrency inherent in  $\lambda$ -calculus terms, rather than simulating standard reduction. Wadler [22] adapts the approach of Caires and Pfenning to classical (rather than intuitionistic) linear logic, and gives a translation from GV (his functional language) to CP (his process calculus). He does not give a direct semantics for GV. In previous work [16], we give a translation between CP and GV.

**Deadlock Freedom and Progress.** There have been several approaches to guarantee deadlock freedom in  $\pi$ -calculus. Kobayashi [13] and Padovani [17] give type systems that guarantee deadlock freedom. Their approaches capturing priority information in types, restricting the order in which channels are used, and thus statically eliminating the possibility of cyclic dependencies. Their approaches can be seen as developments of linear  $\pi$ -calculus—they do not directly address the evolution of types provided by session types.

Carbone and Debois [6] give a graphical characterisation of session-typed processes; this allows them to directly identify cycles in the session interactions. They show that all possible interactions eventually take place in cycle-free processes. Carbone et al. [5] show similar results for well-typed processes under Kobayashi’s type system for deadlock freedom; their approach accommodates processes with open channels by defining a type-directed closure of a process, and showing that open processes progress only if their typed closures progress.

## 6 Conclusion and Future Work

We have presented a small-step operational semantics for GV, a minimal session-typed functional language. We have shown that it is deadlock-free, deterministic, and terminating, and have established simulations both ways between our semantics for GV and cut-reduction in a process calculus based on linear logic. Finally, we have demonstrated that GV is a suitable basis for future modular language development by illustrating two extensions of GV, and observing that our proof of deadlock freedom extends directly to these cases.

In closing, we identify two important directions for future work. The first direction is recursion. Recursion is essential both for sessions (to capture repeating behavior, such as adding recipients to a mail message) and for functional programming. Adding unchecked recursion to GV would clearly compromise



termination and introduce the possibility of livelock; we hope that adapting approaches used for fixed points in linear logic might mitigate this issue. The second direction is asynchrony. We hope to develop the approach of Gay and Vasconcelos [8] and show a correspondence between synchronous and asynchronous semantics for GV.

## References

- [1] S. Abramsky. Proofs as processes. *Theor. Comput. Sci.*, 135(1):5–9, Apr. 1992.
- [2] H. P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*, volume 103. North Holland, revised edition, 1984.
- [3] G. Bellin and P. J. Scott. On the  $\pi$ -Calculus and linear logic. *Theoretical Computer Science*, 135(1):11–65, 1994.
- [4] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, pages 222–236, 2010.
- [5] M. Carbone, O. Dardha, and F. Montesi. Progress as compositional lock-freedom. In *COORDINATION 2014*, pages 49–64, 2014.
- [6] M. Carbone and S. Debois. A graphical approach to progress for structured communication in web services. In *ICE*, pages 13–27, 2010.
- [7] O. Dardha, E. Giachino, and D. Sangiorgi. Session types revisited. In *PPDP*, pages 139–150, 2012.
- [8] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(01):19–50, 2010.
- [9] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, Jan. 1987.
- [10] K. Honda. Types for dyadic interaction. In *CONCUR*, pages 509–523, 1993.
- [11] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, pages 122–138, 1998.
- [12] N. Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, pages 439–453, 2002.
- [13] N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR*, pages 233–247, 2006.
- [14] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. In *POPL*, pages 358–371, 1996.
- [15] J. Lévy and L. Maranget. Explicit substitutions and programming languages. In *Foundations of Software Technology and Theoretical Computer Science, 1999*, volume 1738 of *LNCS*, pages 181–200. Springer, 1999.
- [16] S. Lindley and J. G. Morris. Sessions as propositions. In *PLACES*, 2014.
- [17] L. Padovani. Deadlock and lock freedom in the linear  $\pi$ -calculus. In *LICS*, page 72, 2014.
- [18] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE*, pages 398–413, 1994.
- [19] B. Toninho, L. Caires, and F. Pfenning. Functions as session-typed processes. In *FOSSACS*, pages 346–360, 2012.
- [20] S. van Bakel and M. G. Vigliotti. A logical interpretation of the  $\lambda$ -calculus into the  $\pi$ -calculus, preserving spine reduction and types. In *CONCUR*, pages 84–98, 2009.
- [21] V. T. Vasconcelos, S. J. Gay, and A. Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.
- [22] P. Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.

## A A Simpler Send

The CP send rule 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  $Q$ . This complicates both the reduction semantics of CP (as the cut reduction of  $\otimes$  against  $\wp$  must account for all three behaviours) and the equivalence of CP and GV (where the behavior of **send** is simpler).

We can give an alternative formulation of send, avoiding the additional name restriction and composition, as follows:

$$\frac{P \vdash \Delta, x : B, y : A}{x\langle y \rangle.P \vdash \Delta, x : A \otimes B, y : A^\perp}$$

where  $x\langle y \rangle.P$  is defined as  $x[z].(y \leftrightarrow z \mid P)$ . In particular, note that

$$\begin{aligned} \nu x (x\langle y \rangle.P \mid x(z).Q) &= \nu x (x[z].(y \leftrightarrow z \mid P) \mid x(z).Q) \\ &\longrightarrow_C \nu z (y \leftrightarrow z \mid \nu x (P \mid Q)) \\ &\longrightarrow_C \nu x (P \mid Q\{y/z\}) \end{aligned}$$

as we would expect for synchronising a send and a receive. Similarly, we note that any process  $x[y].(P \mid Q)$  can also be expressed as a process  $\nu y (P \mid x\langle y \rangle.Q)$ , which reduces to the original by one application of the commuting conversions. However, the two formulations are not quite identical. Let us consider the possible reductions of the two terms. Notice that in  $x[y].(P \mid Q)$ , both  $P$  and  $Q$  are blocked on  $x$ ; however, the same is not true for  $\nu y (P \mid x\langle y \rangle.Q)$ ; the latter permits reductions in  $P$  before synchronising on  $x$ .

## B Proofs of Selected Results

**Theorem 2.** If  $\Gamma \vdash C$  and  $C \longrightarrow C'$  then there is some  $\Gamma'$  such that  $\Gamma \longrightarrow^? \Gamma'$  and  $\Gamma' \vdash C'$ .

*Proof.* By induction on the derivation of  $C \longrightarrow C'$ . We include several representative cases.

- Case LIFT is immediate by the induction hypothesis.
- Case LIFTV follows from Lemma 1.
- In case SEND, from the assumption  $\Gamma \vdash F[\text{send}(V, x)] \parallel F'[\text{receive } x]$ , we can assume that  $\Gamma$  partitions as  $\Gamma_1, \Gamma_2, x : S^\sharp$  such that  $S = !T.S'$ ,  $V$  has type  $T$ , and  $\Gamma_1, x : !T.S' \vdash F[\text{send}(V, x)]$ ,  $\Gamma_2, x : ?T.\bar{S}' \vdash F'[\text{receive } x]$ . As  $\text{send} : T \times !T.S' \multimap S'$  and  $\Gamma_1, x : !T.S' \vdash F[\text{send}(V, x)]$ , we can conclude that  $\Gamma_1, x : S' \vdash F[x]$ . By a similar argument, we conclude that  $\Gamma_2, x : \bar{S}' \vdash F'[(V, x)]$ . Finally, as  $(!T.S)^\sharp \longrightarrow S^\sharp$ ,  $\Gamma_1, \Gamma_2, x : (!T.S)^\sharp \longrightarrow \Gamma_1, \Gamma_2, x : S^\sharp$ , and we can recompose the resulting processes concluding that  $\Gamma_1, \Gamma_2, x : S^\sharp \vdash F[x] \parallel F'[(V, x)]$ .

- In case **FORK**, from the assumption  $\Gamma \vdash F[\text{fork } V]$ , we can conclude that  $\Gamma$  splits as  $\Gamma_1, \Gamma_2$  and there is some  $S$  such that  $\Gamma_1, x : S \vdash F[x]$  and  $\Gamma_2, x : \bar{S} \vdash V x$ . Thus we have that  $\Gamma, x : S \vdash F[x] \parallel V x$  and  $\Gamma \vdash (\nu x)(F[x] \parallel V x)$ .
- In case **WAIT**, from the assumption  $\Gamma \vdash (\nu x)(F[\text{wait } x] \parallel x)$ , we can conclude that  $\Gamma, x : \text{end}_? \vdash F[\text{wait } x]$  and thus, from the typing of **wait**, that  $\Gamma \vdash F[()]$ .
- In case **LINK**, from the assumptions  $\Gamma \vdash (\nu x)(F[\text{link}(x, y)] \parallel F'[M]), x \in \text{fv}(M)$ , we can conclude that  $\Gamma$  partitions as  $\Gamma_1, \Gamma_2, y : S$  such that  $\Gamma_1, y : S, x : \bar{S} \vdash F[\text{link}(x, y)]$  and  $\Gamma_2, x : S \vdash F'[M]$ . (Note that the free variable assumption on the reduction rule for **fork** allows us to assume that neither  $F$  nor  $F'$  binds  $x$  or  $y$ .) From the type of **link**, we have that  $\Gamma, x : \text{end}_! \vdash F[x]$ ; similarly, from  $x \in \text{fv}(M)$ , we can conclude that  $\Gamma_2, y : S \vdash F'[M\{y/x\}]$ . Finally, from the typing rule for **wait**, we have that  $\Gamma_2, x : \text{end}_?, y : S \vdash F'[\text{wait } x; M\{y/x\}]$ , and that  $\Gamma \vdash (\nu x)(F[x] \parallel F'[\text{wait } x; M\{y/x\}])$   $\square$

**Lemma 9.** If  $\Gamma \vdash C$ , then there is some  $C' \equiv C$  such that  $\Gamma \vdash C'$  and  $C'$  is in canonical form.

*Proof.* Let  $x_1, \dots, x_{n-1}$  be the  $\nu$ -bound variables in  $C$  and  $M_1, \dots, M_n$  be the terms in  $P$ ; the proof is by induction on  $n$ . If  $n > 1$ , then pick some  $M_i$  such that there is exactly one  $\nu$ -bound variable  $x_j$  where  $x_j \in \text{fv}(M_i)$ . (That there must be such an  $M_i$  and  $x_j$  can be established by a standard counting argument, together with Lemma 5.) Now, construct  $D$  from  $C$  by the homomorphic extension of the mapping  $(\nu x_j)E \mapsto E; E \parallel \phi M_i \mapsto E$ . From the assumption that  $\Gamma \vdash C$ , we can conclude that there is some  $\Gamma' \subseteq \Gamma$  and type  $S$  such that  $\Gamma', x_j : S \vdash D$ . By the induction hypothesis, there is some  $D' \equiv D$  in canonical form. Finally, let  $C' = (\nu x_j)(\phi M_i \parallel D')$ ; we can see that straightforwardly that  $C'$  is in canonical form; that  $C \equiv C'$ ; and, that  $\Gamma \vdash C'$ .  $\square$

**Theorem 22.** If  $\Gamma \vdash C$ ,  $\Gamma' \vdash D$ ,  $\Gamma \longrightarrow \Gamma'$ , and  $C \longrightarrow D$ , then  $\llbracket C \rrbracket z \Longrightarrow \llbracket D \rrbracket z$ .

*Proof.* By induction on the derivation of  $C \longrightarrow D$ . The inductive cases follow straightforwardly from the compositionality of the definitions and Theorem 21. The case for **send** against **receive** is included in the main body of the paper. The remaining cases follow.

- Case  $(\nu x)(F[\text{wait } x] \parallel \circ x) \longrightarrow F[()]$ .

$$\begin{aligned}
& \llbracket (\nu x)(F[\text{wait } x] \parallel \circ x) \rrbracket z \\
&= \\
& \quad \nu x (\llbracket F \rrbracket [y \mapsto \nu w (w \leftrightarrow x \mid w \leftrightarrow y)] z \mid \nu y (x \leftrightarrow y \mid y[])) \\
& \Longrightarrow_C (\text{cut links}) \\
& \quad \nu x (\llbracket F \rrbracket [y \mapsto x \leftrightarrow y] z \mid x[]) \\
& \Longrightarrow_C (\text{cut link}) \\
& \quad \llbracket F \rrbracket [y \mapsto y[]] z \\
&= \\
& \quad \llbracket F[()] \rrbracket z
\end{aligned}$$

– Case  $F[\text{fork}(\lambda^\sigma x.M)] \longrightarrow (\nu x)(F[x] \parallel M\sigma)$ .

$$\begin{aligned}
& \llbracket F[\text{fork}(\lambda^\sigma x.M)] \rrbracket z \\
&= \\
& \llbracket F \rrbracket [\llbracket \text{fork}(\lambda^\sigma x.M) \rrbracket] z \\
&= \\
& \llbracket F \rrbracket [z \mapsto \nu x (\llbracket x \rrbracket z \mid \nu y (\llbracket \sigma \rrbracket (y(x). \llbracket M \rrbracket y) \mid \nu w (y \langle x \rangle . y \leftrightarrow w \mid w \llbracket \rrbracket))) ] z \\
& \Longrightarrow_C \text{ (cut send against receive)} \\
& \llbracket F \rrbracket [z \mapsto \nu x (\llbracket x \rrbracket z \mid \nu y (\llbracket \sigma \rrbracket (\llbracket M \rrbracket y) \mid y \llbracket \rrbracket))] z \\
& \equiv \\
& \nu x (\llbracket F \rrbracket [\llbracket x \rrbracket] z \mid \llbracket \sigma \rrbracket (\nu y (\llbracket M \rrbracket y \mid y \llbracket \rrbracket))) \\
&= \\
& \llbracket (\nu x)(F[x] \parallel M\sigma) \rrbracket z
\end{aligned}$$

– Case  $(\nu x)(F[\text{link}(x, y)] \parallel F'[M]) \longrightarrow (\nu x)(F[x] \parallel F'[\text{wait } x; M\{y/x\}])$ .

$$\begin{aligned}
& \llbracket (\nu x)(F[\text{link}(x, y)] \parallel F'[M]) \rrbracket z \\
&= \\
& \nu x (\llbracket F \rrbracket [\llbracket \text{link}(x, y) \rrbracket] z \mid \llbracket F' \rrbracket [\llbracket M \rrbracket] z) \\
&= \\
& \nu x (\llbracket F \rrbracket [z \mapsto \nu v (v \leftrightarrow z \mid \nu w (v \leftrightarrow w \mid \\
& \quad \nu x' (x \leftrightarrow x' \mid \nu y' (y \leftrightarrow y' \mid w().x' \leftrightarrow y'))))] z \mid \llbracket F' \rrbracket [\llbracket M \rrbracket] z) \\
& \Longrightarrow_C \text{ (cut links)} \\
& \nu x (\llbracket F \rrbracket [z \mapsto \nu v (v \leftrightarrow z \mid \nu w (v \leftrightarrow w \mid w().x \leftrightarrow y))] z \mid \llbracket F' \rrbracket [\llbracket M \rrbracket] z) \\
& \equiv \\
& \nu v (\llbracket F \rrbracket [z \mapsto v \leftrightarrow z] z \mid \nu x (\nu w (v \leftrightarrow w \mid w().x \leftrightarrow y) \mid \llbracket F' \rrbracket [\llbracket M \rrbracket] z)) \\
& \equiv \\
& \nu v (\llbracket F \rrbracket [z \mapsto v \leftrightarrow z] z \mid \llbracket F' \rrbracket [z \mapsto \nu w (v \leftrightarrow w \mid w().\nu x (x \leftrightarrow y \mid \llbracket M \rrbracket z))] z) \\
&= \\
& \llbracket (\nu x)(F[x] \parallel F'[\text{wait } x; M\{y/x\}]) \rrbracket z
\end{aligned}$$