Prioritise the Best Variation

Wen Kokke, Ornela Dardha

Prioritise the Best Variation*

Wen Kokke[1] and Ornela Dardha[2]

University of Edinburgh, Edinburgh wen.kokke@ed.ac.uk
University of Glasgow, Glasgow ornela.dardha@glasgow.ac.uk

Abstract. Binary session types guarantee communication safety and session fidelity, but *alone* they cannot rule out deadlocks arising from the interleaving of different sessions. In Classical Processes (CP) [53]—a process calculus based on classical linear logic—deadlock freedom is guaranteed by combining channel creation and parallel composition under the same logical cut rule. Similarly, in Good Variation (GV) [54,39]—a linear concurrent λ-calculus—deadlock freedom is guaranteed by combining channel creation and thread spawning under the same operation, called fork. In both CP and GV, deadlock freedom is achieved at the expense of expressivity, as the only processes allowed are tree-structured. Dardha and Gay [19] define Priority CP (PCP), which allows cyclic-structured processes and restores deadlock freedom by using *priorities*, in line with Kobayashi and Padovani [34,44]. Following PCP, we present Priority GV (PGV), a variant of GV which decouples channel creation from thread spawning. Consequently, we type cyclic-structured processes and restore deadlock freedom by using priorities. We show that our type system is sound by proving subject reduction and progress. We define an encoding from PCP to PGV and prove that the encoding preserves typing and is sound and complete with respect to the operational semantics.

Keywords: session types · π-calculus · functional programming · deadlock freedom · GV · CP

1 Introduction

Session types [29,47,30] are types for protocols. Regular types ensure functions are used according to their specification. Session types ensure *communication channels* are used according to their protocols. Session types have been studied in many settings. For instance, in the π-calculus [29,47,30], a foundational calculus for communication and concurrency, and in concurrent λ-calculi [26], including the focus of our paper: Good Variation [54,39, GV].

GV is a concurrent λ-calculus with *binary* session types, where each channel is shared between exactly two processes. Binary session types guarantee two crucial properties *communication safety*—*e.g.*, if the protocol says to transmit an integer, you transmit an integer—and *session fidelity*—*e.g.*, if the protocol says send, you send. A third crucial property is *deadlock freedom*, which ensures

* Supported by the EU HORIZON 2020 MSCA RISE project 778233 "Behavioural Application Program Interfaces" (BehAPI).

that processes do not have cyclic dependencies—*e.g.*, when two processes wait for each other to send a value. Binary session types *alone* are insufficient to rule out deadlocks arising from interleaved sessions, but several additional techniques have been developed to guarantee deadlock freedom in session-typed π-calculus and concurrent λ-calculus.

In the π-calculus literature, there have been several attempts at developing Curry-Howard correspondences between session-typed π-calculus and linear logic [27]: Caires and Pfenning's πDILL [9] corresponds to dual intuitionistic linear logic [4], and Wadler's Classical Processes [53, CP] corresponds to classical linear logic [27, CLL]. Both calculi guarantee deadlock freedom, which they achieve by restricting structure of processes and shared channels to *trees*, by combing name restriction and parallel composition into a single construct, corresponding to the logical cut. This ensures that two processes can only communicate via exactly one series of channels, which rules out interleavings of sessions, and guarantees deadlock freedom. There are many downsides to combining name restriction and parallel composition, such as lack of modularity, difficulty typing structural congruence and formulating label-transition semantics, which have led to various approaches to decoupling these constructs. Hypersequent CP [41,37,38] and Linear Compositional Choreographies [14] decouple them, but maintain the correspondence to CLL and allow only tree-structured processes. Priority CP [20, PCP] weakens the correspondence to CLL in exchange for a more expressive language which allows cyclic-structured processes. PCP decouples CP's cut rule into two separate constructs: one for parallel composition via a mix rule, and one for name restriction via a cycle rule. To restore deadlock freedom, PCP uses *priorities* [34,44]. Priorities encode the *order of actions* and rule out bad cyclic interleavings. Dardha and Gay [20] prove cycle-elimination for PCP, adapting the cut-elimination proof for classical linear logic, and deadlock freedom follows as a corollary.

CP and GV are related via a pair of translations which satisfy simulation [40], and which can be tweaked to satisfy reflection. The two calculi share the same strong guarantees. GV achieves deadlock freedom via a similar syntactic restriction: it combines channel creation and thread spawning into a single operation, called "fork", which is related to the cut construct in CP. Unfortunately, as with CP, this syntactic restriction has its downsides.

Our aim is to develop a more expressive version of GV while maintaining deadlock freedom. While process calculi have their advantages, *e.g.*, their succinctness, we chose to work with GV for several reasons. In general, concurrent λ-calculi support higher-order functions, and have a capability for abstraction not usually present in process calculi. Within a concurrent λ-calculus, one can derive extensions of the communication capabilities of the language via well-understood extensions of the functional fragment, *e.g.*, we can derive internal/external choice from sum types. Concurrent λ-calculi maintain a clear separation between the program which the user writes and the configurations which represent the state of the system as it evaluates the program. However, our main motivation is that results obtained for λ-calculi transfer more easily to real-world functional pro-

gramming languages. Case in point: we easily adapted the type system of PGV to Linear Haskell [6], which gives us a library for deadlock-free session-typed programming [36]. The benefit of working specifically with GV, as opposed to other concurrent λ-calculi, is its relation to CP [53], and its formal properties, including deadlock freedom. We thus pose our research question for GV:

> **RQ:** Can we design a more expressive GV which guarantees deadlock freedom for cyclic-structured processes?

We follow the line of work from CP to Priority CP, and present Priority GV (PGV), a variant of GV which decouples channel creation from thread spawning, thus allowing cyclic-structured processes, but which nonetheless guarantees deadlock freedom via priorities. This closes the circle of the connection between CP and GV [53], and their priority-based versions, PCP [20] and PGV. We make the following main contributions:

(§2) **Priority GV**. We present Priority GV (§2, PGV), a session-typed functional language with priorities, and prove subject reduction (theorem 1) and progress (theorem 2). We addresses several problems in the original GV language, most notably: (a) PGV does not require the pseudo-type $S^\sharp$; and (b) its structural congruence is type preserving. PGV answers our research question positively as it allows cyclic-structured binary session-typed processes that are deadlock free.

(§3) **Translation from PCP to PGV**. We present a *sound and complete encoding* of Priority CP [20] in PGV (§3). We prove the encoding preserves typing (theorem 4) and satisfies operational correspondence (theorems 5 and 6). To obtain a tight correspondence, we update PCP, moving away from commuting conversions and reduction as cut elimination towards reduction based on structural congruence, as it is standard in process calculi.

2 Priority GV

We present Priority GV (PGV), a session-typed functional language based on GV [54,39] which uses priorities à la Kobayashi and Padovani [34,45] to enforce deadlock freedom. Priority GV offers a more fine-grained analysis of communication structures, and by separating channel creation form thread spawning it allows cyclic structures. We illustrate this with two programs in PGV, examples 1 and 2. Each program contains two processes—the main process, and the child process created by **spawn**—which communicate using *two* channels. The child process receives a unit over the channel x/x', and then sends a unit over the channel y/y'. The main process does one of two things: (a) in example 1, it sends a unit over the channel x/x', and then waits to receive a unit over the channel y/y'; (b) in example 2, it does these in the opposite order, which results in a deadlock. PGV is more expressive than GV: example 1 is typeable and guaranteed deadlock-free in PGV, but is not typeable in GV [53] and not guaranteed deadlock-free in GV's predecessor [26]. We believe PGV is a non-conservative *extension* of GV, as CP can be embedded in a Kobayashi-style system [22].

Example 1 (Cyclic Structure).

$$\begin{array}{l}\textbf{let } (x, x') = \textbf{new in}\\ \textbf{let } (y, y') = \textbf{new in}\\ \textbf{spawn}\left(\begin{array}{l}\textbf{let } ((), x') = \textbf{recv } x' \textbf{ in}\\ \textbf{let } y = \textbf{send } ((), y) \textbf{ in}\\ \textbf{wait } x'; \textbf{close } y\end{array}\right);\\ \underline{\textbf{let } x = \textbf{send } ((), x) \textbf{ in}}\\ \underline{\textbf{let } ((), y') = \textbf{recv } y' \textbf{ in}}\\ \textbf{close } x; \textbf{wait } y'\end{array}$$

Example 2 (Deadlock).

$$\begin{array}{l}\textbf{let } (x, x') = \textbf{new in}\\ \textbf{let } (y, y') = \textbf{new in}\\ \textbf{spawn}\left(\begin{array}{l}\textbf{let } ((), x') = \textbf{recv } x' \textbf{ in}\\ \textbf{let } y = \textbf{send } ((), y) \textbf{ in}\\ \textbf{wait } x'; \textbf{close } y\end{array}\right);\\ \underline{\textbf{let } ((), y') = \textbf{recv } y' \textbf{ in}}\\ \underline{\textbf{let } x = \textbf{send } ((), x) \textbf{ in}}\\ \textbf{close } x; \textbf{wait } y'\end{array}$$

Session types. Session types (S) are defined by the following grammar:

$$S ::= !^{o}T.S \mid ?^{o}T.S \mid \mathbf{end}_{!}^{o} \mid \mathbf{end}_{?}^{o}$$

Session types $!^{o}T.S$ and $?^{o}T.S$ describe the endpoints of a channel over which we send or receive a value of type T, and then proceed as S. Types $\mathbf{end}_{!}^{o}$ and $\mathbf{end}_{?}^{o}$ describe endpoints of a channel whose communication has finished, and over which we must synchronise before closing the channel. Each connective in a session type is annotated with a *priority* $o \in \mathbb{N}$.

Types. Types (T, U) are defined by the following grammar:

$$T, U ::= T \times U \mid \mathbf{1} \mid T + U \mid \mathbf{0} \mid T \multimap^{p,q} U \mid S$$

Types $T \times U$, $\mathbf{1}$, $T + U$, and $\mathbf{0}$ are the standard linear λ-calculus product type, unit type, sum type, and empty type. Type $T \multimap^{p,q} U$ is the standard linear function type, annotated with *priority bounds* $p, q \in \mathbb{N} \cup \{\bot, \top\}$. Every session type is also a type. Given a function with type $T \multimap^{p,q} U$, p is a *lower bound* on the priorities of the endpoints captured by the body of the function, and q is an *upper bound* on the priority of the communications that take place as a result of applying the function. The type of *pure functions* $T \multimap U$, *i.e.*, those which perform no communications, is syntactic sugar for $T \multimap^{\top,\bot} U$.

Environments. Typing environments Γ, Δ associate types to names. Environments are linear, so two environments can only be combined as Γ, Δ if their names are distinct, *i.e.*, $\mathrm{fv}(\Gamma) \cap \mathrm{fv}(\Delta) = \varnothing$.

$$\Gamma, \Delta ::= \varnothing \mid \Gamma, x : T$$

Duality. Duality plays a crucial role in session types. The two endpoints of a channel are assigned dual types, ensuring that, for instance, whenever one program *sends* a value on a channel, the program on the other end is waiting to *receive*. Each session type S has a dual, written $\overline{S}$. Duality is an involutive function which *preserves priorities*:

$$\overline{!^{o}T.S} = ?^{o}T.\overline{S} \qquad \overline{?^{o}T.S} = !^{o}T.\overline{S} \qquad \overline{\mathbf{end}_{!}^{o}} = \mathbf{end}_{?}^{o} \qquad \overline{\mathbf{end}_{?}^{o}} = \mathbf{end}_{!}^{o}$$

Priorities. Function $\mathrm{pr}(\cdot)$ returns the smallest priority of a session type. The type system guarantees that the top-most connective always holds the smallest priority, so we simply return the priority of the top-most connective:

$$\mathrm{pr}(!^o T.S) = o \qquad \mathrm{pr}(?^o T.S) = o \qquad \mathrm{pr}(\mathbf{end}_!^o) = o \qquad \mathrm{pr}(\mathbf{end}_?^o) = o$$

We extend the function $\mathrm{pr}(\cdot)$ to types and typing contexts by returning the smallest priority in the type or context, or $\top$ if there is no priority. We use $\sqcap$ and $\sqcup$ to denote the minimum and maximum:

$$\begin{array}{lll} \mathrm{min}_{\mathrm{pr}}(T \times U) & = \mathrm{min}_{\mathrm{pr}}(T) \sqcap \mathrm{min}_{\mathrm{pr}}(U) & \mathrm{min}_{\mathrm{pr}}(\mathbf{1}) = \top \\ \mathrm{min}_{\mathrm{pr}}(T + U) & = \mathrm{min}_{\mathrm{pr}}(T) \sqcap \mathrm{min}_{\mathrm{pr}}(U) & \mathrm{min}_{\mathrm{pr}}(\mathbf{0}) = \top \\ \mathrm{min}_{\mathrm{pr}}(T \multimap^{p,q} U) & = p & \mathrm{min}_{\mathrm{pr}}(S) = \mathrm{pr}(S) \\ \mathrm{min}_{\mathrm{pr}}(\Gamma, x : A) & = \mathrm{min}_{\mathrm{pr}}(\Gamma) \sqcap \mathrm{min}_{\mathrm{pr}}(A) & \mathrm{min}_{\mathrm{pr}}(\varnothing) = \top \end{array}$$

Terms. Terms (L, M, N) are defined by the following grammar:

$$\begin{array}{lll} L, M, N & ::= & x \mid K \mid \lambda x.M \mid M\ N \\ & & \mid () \mid M; N \mid (M, N) \mid \mathbf{let}\ (x, y) = M\ \mathbf{in}\ N \\ & & \mid \mathbf{inl}\ M \mid \mathbf{inr}\ M \mid \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} \mid \mathbf{absurd}\ M \\ K & ::= & \mathbf{link} \mid \mathbf{new} \mid \mathbf{spawn} \mid \mathbf{send} \mid \mathbf{recv} \mid \mathbf{close} \mid \mathbf{wait} \end{array}$$

Let x, y, z, and w range over variable names. Occasionally, we use a, b, c, and d. The term language is the standard linear λ-calculus with products, sums, and their units, extended with constants K for the communication primitives.

Constants are best understood in conjunction with their typing and reduction rules in figs. 1 and 2. Briefly, **link** links two endpoints together, forwarding messages from one to the other, **new** creates a new channel and returns a pair of its endpoints, and **spawn** spawns off its argument as a new thread. The **send** and **recv** functions send and receive values on a channel. However, since the typing rules for PGV ensure the linear usage of endpoints, they also return a new copy of the endpoint to continue the session. The **close** and **wait** functions close a channel. We use syntactic sugar to make terms more readable: we write $\mathbf{let}\ x = M\ \mathbf{in}\ N$ in place of $(\lambda x.N)\ M$, $\lambda().M$ in place of $\lambda z.z; M$, and $\lambda(x, y).M$ in place of $\lambda z.\mathbf{let}\ (x, y) = z\ \mathbf{in}\ M$. We recover **fork** as $\lambda x.\mathbf{let}\ (y, z) = \mathbf{new}\ ()\ \mathbf{in}\ \mathbf{spawn}\ (\lambda().x\ y); z$.

Internal and External Choice. Typically, session-typed languages feature constructs for internal and external choice. In GV, these can be defined in terms of the core language, by sending or receiving a value of a sum type [39]. We use the following syntactic sugar for internal ($S \oplus^o S'$) and external ($S \mathbin{\&}^o S'$) choice and their units:

$$\begin{array}{ll} S \oplus^o S' \triangleq !^o(\overline{S} + \overline{S'}).\mathbf{end}_!^{o+1} & \oplus^o\{\} \triangleq !^o\mathbf{0}.\mathbf{end}_!^{o+1} \\ S \mathbin{\&}^o S' \triangleq ?^o(S + S').\mathbf{end}_?^{o+1} & \mathbin{\&}^o\{\} \triangleq ?^o\mathbf{0}.\mathbf{end}_?^{o+1} \end{array}$$

As the syntax for units suggests, these are the binary and nullary forms of the more common n-ary choice constructs $\oplus^o\{l_i : S_i\}_{i \in I}$ and $\mathbin{\&}^o\{l_i : S_i\}_{i \in I}$, which

one may obtain generalising the sum types to variant types. For simplicity, we present only the binary and nullary forms.

Similarly, we use syntactic sugar for the term forms of choice, which combine sending and receiving with the introduction and elimination forms for the sum and empty types. There are two constructs for binary internal choice, expressed using the meta-variable ℓ which ranges over $\{\mathbf{inl}, \mathbf{inr}\}$. As there is no introduction for the empty type, there is no construct for nullary internal choice:

$$\begin{aligned}
&\mathbf{select}\ \ell \quad \triangleq \lambda x.\mathbf{let}\ (y,z) = \mathbf{new}\ \mathbf{in}\ \mathbf{close}\ (\mathbf{send}\ (\ell\ y, x)); z \\
&\mathbf{offer}\ L\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} \triangleq \\
&\quad \mathbf{let}\ (z,w) = \mathbf{recv}\ L\ \mathbf{in}\ \mathbf{wait}\ w; \mathbf{case}\ z\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} \\
&\mathbf{offer}\ L\ \{\} \triangleq \mathbf{let}\ (z,w) = \mathbf{recv}\ L\ \mathbf{in}\ \mathbf{wait}\ w; \mathbf{absurd}\ z
\end{aligned}$$

Operational Semantics. Priority GV terms are evaluated as part of a configuration of processes. Configurations are defined by the following grammar:

$$\phi ::= \bullet \mid \circ \qquad\qquad \mathcal{C}, \mathcal{D}, \mathcal{E} ::= \phi\ M \mid \mathcal{C} \parallel \mathcal{D} \mid (\nu x x')\mathcal{C}$$

Configurations ($\mathcal{C}$, $\mathcal{D}$, $\mathcal{E}$) consist of threads $\phi\ M$, parallel compositions $\mathcal{C} \parallel \mathcal{D}$, and name restrictions $(\nu x x')\mathcal{C}$. To preserve the functional nature of PGV, where programs return a single value, we use flags (ϕ) to differentiate between the main thread, marked $\bullet$, and child threads created by **spawn**, marked $\circ$. Only the main thread returns a value. We determine the flag of a configuration by combining the flags of all threads in that configuration:

$$\bullet + \circ = \bullet \qquad \circ + \bullet = \bullet \qquad \circ + \circ = \circ \qquad (\bullet + \bullet \text{ is undefined})$$

The use of $\circ$ for child threads [39] overlaps with the use of the meta-variable o for priorities [20]. Both are used to annotate sequents: flags appear on the sequent in configuration typing, and priorities in term typing. To distinguish the two symbols, they are typeset in a different font and a different colour.

Values (V, W), evaluation contexts (E), thread evaluation contexts ($\mathcal{F}$), and configuration contexts ($\mathcal{G}$) are defined by the following grammars:

$$\begin{aligned}
V, W &::= x \mid K \mid \lambda x.M \mid () \mid (V, W) \mid \mathbf{inl}\ V \mid \mathbf{inr}\ V \\
E &::= \Box \mid E\ M \mid V\ E \\
&\quad \mid E; N \mid (E, M) \mid (V, E) \mid \mathbf{let}\ (x,y) = E\ \mathbf{in}\ M \\
&\quad \mid \mathbf{inl}\ E \mid \mathbf{inr}\ E \mid \mathbf{case}\ E\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} \mid \mathbf{absurd}\ E \\
\mathcal{F} &::= \phi\ E \\
\mathcal{G} &::= \Box \mid \mathcal{G} \parallel \mathcal{C} \mid (\nu x y)\mathcal{G}
\end{aligned}$$

We factor the reduction relation of PGV into a deterministic reduction on terms ($\longrightarrow_M$) and a non-deterministic reduction on configurations ($\longrightarrow_{\mathcal{C}}$), see fig. 1. We write $\longrightarrow_M^+$ and $\longrightarrow_{\mathcal{C}}^+$ for the transitive closures, and $\longrightarrow_M^\star$ and $\longrightarrow_{\mathcal{C}}^\star$ for the reflexive-transitive closures.

Term reduction is the standard call-by-value, left-to-right evaluation for GV, and only deviates from reduction for the linear λ-calculus in that it reduces terms to values *or* ready terms waiting to perform a communication action.

Term reduction.

$$
\begin{array}{lll}
\text{E-Lam} & (\lambda x.M)\ V & \longrightarrow_M M\{V/x\} \\
\text{E-Unit} & \mathbf{let}\ () = ()\ \mathbf{in}\ M & \longrightarrow_M M \\
\text{E-Pair} & \mathbf{let}\ (x, y) = (V, W)\ \mathbf{in}\ M & \longrightarrow_M M\{V/x\}\{W/y\} \\
\text{E-Inl} & \mathbf{case}\ \mathbf{inl}\ V\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} & \longrightarrow_M M\{V/x\} \\
\text{E-Inr} & \mathbf{case}\ \mathbf{inr}\ V\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} & \longrightarrow_M N\{V/y\}
\end{array}
$$

$$
\text{E-Lift}\ \frac{M \longrightarrow_M M'}{E[M] \longrightarrow_M E[M']}
$$

Structural congruence.

$$
\begin{array}{lll}
\text{SC-LinkSwap} & \mathcal{F}[\mathbf{link}\ (x, y)] & \equiv \mathcal{F}[\mathbf{link}\ (y, x)] \\
\text{SC-ResLink} & (\nu xy)(\phi\ \mathbf{link}\ (x, y)) & \equiv \phi\ () \\
\text{SC-ResSwap} & (\nu xy)\mathcal{C} & \equiv (\nu yx)\mathcal{C} \\
\text{SC-ResComm} & (\nu xy)(\nu zw)\mathcal{C} & \equiv (\nu zw)(\nu xy)\mathcal{C},\ \text{if}\ \{x, y\} \cap \{z, w\} = \varnothing \\
\text{SC-ResExt} & (\nu xy)(\mathcal{C} \parallel \mathcal{D}) & \equiv \mathcal{C} \parallel (\nu xy)\mathcal{D},\ \text{if}\ x, y \notin \text{fv}(\mathcal{C}) \\
\text{SC-ParNil} & \mathcal{C} \parallel \circ() & \equiv \mathcal{C} \\
\text{SC-ParComm} & \mathcal{C} \parallel \mathcal{D} & \equiv \mathcal{D} \parallel \mathcal{C} \\
\text{SC-ParAssoc} & \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) & \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}
\end{array}
$$

Configuration reduction.

$$
\begin{array}{lll}
\text{E-Link} & (\nu xy)(\mathcal{F}[\mathbf{link}\ (w, x)] \parallel \mathcal{C}) & \longrightarrow_{\mathcal{C}} \mathcal{F}[()] \parallel \mathcal{C}\{w/y\} \\
\text{E-New} & \mathcal{F}[\mathbf{new}\ ()] & \longrightarrow_{\mathcal{C}} (\nu xy)(\mathcal{F}[(x, y)]),\ \text{if}\ x, y \notin \text{fv}(\mathcal{F}) \\
\text{E-Spawn} & \mathcal{F}[(\mathbf{spawn}\ V)] & \longrightarrow_{\mathcal{C}} \mathcal{F}[()] \parallel \circ\ V\ () \\
\text{E-Send} & (\nu xy)(\mathcal{F}[\mathbf{send}\ (V, x)] \parallel \mathcal{F}'[\mathbf{recv}\ y]) & \longrightarrow_{\mathcal{C}} (\nu xy)(\mathcal{F}[x] \parallel \mathcal{F}'[(V, y)]) \\
\text{E-Close} & (\nu xy)(\mathcal{F}[\mathbf{wait}\ x] \parallel \mathcal{F}'[\mathbf{close}\ y]) & \longrightarrow_{\mathcal{C}} \mathcal{F}[()] \parallel \mathcal{F}'[()]
\end{array}
$$

$$
\text{E-LiftC}\ \frac{\mathcal{C} \longrightarrow_{\mathcal{C}} \mathcal{C}'}{\mathcal{G}[\mathcal{C}] \longrightarrow_{\mathcal{C}} \mathcal{G}[\mathcal{C}']}
\qquad
\text{E-LiftM}\ \frac{M \longrightarrow_M M'}{\mathcal{F}[M] \longrightarrow_M \mathcal{F}[M']}
\qquad
\text{E-LiftSC}\ \frac{\mathcal{C} \equiv \mathcal{C}' \qquad \mathcal{C}' \longrightarrow_{\mathcal{C}} \mathcal{D}' \qquad \mathcal{D}' \equiv \mathcal{D}}{\mathcal{C} \longrightarrow_{\mathcal{C}} \mathcal{D}}
$$

Fig. 1. Operational Semantics for PGV.

Configuration reduction resembles evaluation for a process calculus: E-Link, E-Send, and E-Close perform communications, E-LiftC allows reduction under configuration contexts, and E-LiftSC embeds a structural congruence $\equiv$. The remaining rules mediate between the process calculus and the functional language: E-New and E-Spawn evaluate the **new** and **spawn** constructs, creating the equivalent configuration constructs, and E-LiftM embeds term reduction.

Structural congruence satisfies the following axioms: SC-LinkSwap allows swapping channels in the link process. SC-ResLink allows restriction to applied to link which is structurally equivalent to the terminated process, thus allowing elimination of unnecessary restrictions. SC-ResSwap allows swapping channels and SC-ResComm states that restriction is commutative. SC-ResExt is the standard scope extrusion rule. Rules SC-ParNil, SC-ParComm and SC-ParAssoc

state that parallel composition uses the terminated process as the neutral element; it is commutative and associative.

While our configuration reduction is based on the standard evaluation for GV, the increased expressiveness of PGV allows us to simplify the relation on two counts. (a) *We decompose the* **fork** *construct.* In GV, **fork** creates a new channel, spawns a child thread, and, when the child thread finishes, it closes the channel to its parent. In PGV, these are three separate operations: **new**, **spawn**, and **close**. We no longer require that every child thread finishes by returning a terminated channel. Consequently, we also simplify the evaluation of the **link** construct. Intuitively, evaluating **link** causes a substitution: if we have a channel bound as (νxy), then **link** (w, x) replaces all occurrences of y by w. However, in GV, **link** is required to return a terminated channel, which means that the semantics for *link* must *create* a fresh channel of type $\mathbf{end}_!/\mathbf{end}_?$. The endpoint of type $\mathbf{end}_!$ is returned by the *link* construct, and a **wait** on the other endpoint guards the *actual* substitution. In PGV, evaluating **link** simply causes a substitution. (b) *Our structural congruence is type preserving.* Consequently, we can embed it directly into the reduction relation. In GV, this is not the case, and subject reduction relies on proving that if $\equiv\!\!\longrightarrow_{\mathcal{C}}$ ends up in an ill-typed configuration, we can rewrite it to a well-typed configuration using $\equiv$.

Typing. Figure 2 gives the typing rules for PGV. Typing rules for terms are at the top of fig. 2. Terms are typed by a judgement $\Gamma \vdash^{p} M : T$ stating that "a term M has type T and an upper bound on its priority p under the typing environment Γ". Typing for the linear λ-calculus is standard. Linearity is ensured by splitting environments on branching rules, requiring that the environment in the variable rule consists of just the variable, and the environment in the constant and unit rules are empty. Constants K are typed using type schemas, and embedded using T-Const (mid of fig. 2). The typing rules treat *all variables* as linear resources, even those of non-linear types such as $\mathbf{1}$. However, the rules can easily be extended to allow values with unrestricted usage [53].

The only non-standard feature of the typing rules is the priority annotations. Priorities are based on *obligations/capabilities* used by Kobayashi [34], and simplified to single priorities following Padovani [44]. The integration of priorities into GV is adapted from Padovani and Novara [45]. Paraphrasing Dardha and Gay [20], priorities obey the following two laws: (i) an action with lower priority happens before an action with higher priority; and (ii) communication requires *equal* priorities for dual actions.

In PGV, we keep track of a lower and upper bound on the priorities of a term, *i.e.*, while evaluating the term, when does it start communicating, and when does it finish. The upper bound is written on the sequent, whereas the lower bound is approximated from the typing environment. Typing rules for sequential constructs enforce sequentially, *e.g.*, the typing for $M; N$ has a side condition which requires that the upper bound of M is smaller than the lower bound of N, *i.e.*, M finishes before N starts. The typing rule for **new** ensures that both endpoints of a channel share the same priorities. Together, these two constraints guarantee deadlock freedom.

Static Typing Rules.

$$\text{T-Var}\ \frac{}{x : T \vdash^{\bot} x : T} \qquad \text{T-Const}\ \frac{}{\varnothing \vdash^{\bot} K : T} \qquad \text{T-Lam}\ \frac{\Gamma, x : T \vdash^{q} M : U}{\Gamma \vdash^{\bot} \lambda x.M : T \multimap^{\min_{\mathrm{pr}}(\Gamma), q} U}$$

$$\text{T-App}\ \frac{\Gamma \vdash^{p} M : T \multimap^{p', q'} U \qquad \Delta \vdash^{q} N : T \qquad p < \min_{\mathrm{pr}}(\Delta) \qquad q < p'}{\Gamma, \Delta \vdash^{p \sqcup q \sqcup q'} M\ N : U}$$

$$\text{T-Unit}\ \frac{}{\varnothing \vdash^{\bot} () : \mathbf{1}} \qquad \text{T-LetUnit}\ \frac{\Gamma \vdash^{p} M : \mathbf{1} \qquad \Delta \vdash^{q} N : T \qquad p < \min_{\mathrm{pr}}(\Delta)}{\Gamma, \Delta \vdash^{p \sqcup q} M; N : T}$$

$$\text{T-Pair}\ \frac{\Gamma \vdash^{p} M : T \qquad \Delta \vdash^{q} N : U \qquad p < \min_{\mathrm{pr}}(\Delta)}{\Gamma, \Delta \vdash^{p \sqcup q} (M, N) : T \times U}$$

$$\text{T-LetPair}\ \frac{\Gamma \vdash^{p} M : T \times T' \qquad \Delta, x : T, y : T' \vdash^{q} N : U \qquad p < \min_{\mathrm{pr}}(\Delta, T, T')}{\Gamma, \Delta \vdash^{p \sqcup q} \mathbf{let}\ (x, y) = M\ \mathbf{in}\ N : U}$$

$$\text{T-Inl}\ \frac{\Gamma \vdash^{p} M : T \qquad \min_{\mathrm{pr}}(T) = \min_{\mathrm{pr}}(U)}{\Gamma \vdash^{p} \mathbf{inl}\ M : T + U} \qquad \text{T-Inr}\ \frac{\Gamma \vdash^{p} M : U \qquad \min_{\mathrm{pr}}(T) = \min_{\mathrm{pr}}(U)}{\Gamma \vdash^{p} \mathbf{inr}\ M : T + U}$$

$$\text{T-CaseSum}\ \frac{\Gamma \vdash^{p} L : T + T' \qquad \Delta, x : T \vdash^{q} M : U \qquad \Delta, y : T' \vdash^{q} N : U \qquad p < \min_{\mathrm{pr}}(\Delta)}{\Gamma, \Delta \vdash^{p \sqcup q} \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} : U}$$

$$\text{T-Absurd}\ \frac{\Gamma \vdash^{p} M : \mathbf{0}}{\Gamma, \Delta \vdash^{p} \mathbf{absurd}\ M : T}$$

Type Schemas for Constants.

$$\mathbf{link} : S \times \overline{S} \multimap \mathbf{1} \qquad \mathbf{new} : \mathbf{1} \multimap S \times \overline{S} \qquad \mathbf{spawn} : (\mathbf{1} \multimap^{p,q} \mathbf{1}) \multimap \mathbf{1}$$

$$\mathbf{send} : T \times !^{o}T.S \multimap^{\top, o} S \qquad \mathbf{recv} : ?^{o}T.S \multimap^{\top, o} T \times S$$

$$\mathbf{close} : \mathbf{end}_{!}^{o} \multimap^{\top, o} \mathbf{1} \qquad \mathbf{wait} : \mathbf{end}_{?}^{o} \multimap^{\top, o} \mathbf{1}$$

Runtime Typing Rules.

$$\text{T-Main}\ \frac{\Gamma \vdash^{p} M : T}{\Gamma \vdash^{\bullet} \bullet\ M} \qquad \text{T-Child}\ \frac{\Gamma \vdash^{p} M : \mathbf{1}}{\Gamma \vdash^{\circ} \circ\ M} \qquad \text{T-Res}\ \frac{\Gamma, x : S, y : \overline{S} \vdash^{\phi} \mathcal{C}}{\Gamma \vdash^{\phi} (\nu xy)\mathcal{C}} \qquad \text{T-Par}\ \frac{\Gamma \vdash^{\phi} \mathcal{C} \qquad \Delta \vdash^{\phi'} \mathcal{D}}{\Gamma, \Delta \vdash^{\phi + \phi'} \mathcal{C} \parallel \mathcal{D}}$$

Fig. 2. Typing Rules for PGV.

To illustrate this, let's go back to the deadlocked program in example 2. Crucially, it composes the terms below in parallel. While each of these terms itself is well-typed, they impose opposite conditions on the priorities, so their composition is ill-typed. (We omit the priorities on $\mathbf{end}_!$ and $\mathbf{end}_?$.)

$$\frac{y' : ?^{o'}\mathbf{1}.\mathbf{end}_? \vdash^{o'} \mathbf{recv}\ y' : \mathbf{1} \times \mathbf{end}_? \qquad x : !^{o}\mathbf{1}.\mathbf{end}_!, y' : \mathbf{end}_? \vdash^{p} \mathbf{let}\ x = \mathbf{send}\ ((), x)\ \mathbf{in}\ \ldots : \mathbf{1} \qquad o' < o}{x : !^{o}\mathbf{1}.\mathbf{end}_!, y' : ?^{o'}\mathbf{1}.\mathbf{end}_? \vdash^{p} \mathbf{let}\ ((), y') = \mathbf{recv}\ y'\ \mathbf{in}\ \mathbf{let}\ x = \mathbf{send}\ ((), x)\ \mathbf{in}\ \ldots : \mathbf{1}}$$

$$\frac{x' : ?^{o}\mathbf{1}.\mathbf{end}_? \vdash^{o} \mathbf{recv}\ x' : \mathbf{1} \times \mathbf{end}_? \qquad y : !^{o'}\mathbf{1}.\mathbf{end}_!, x' : \mathbf{end}_? \vdash^{q} \mathbf{let}\ y = \mathbf{send}\ ((), y)\ \mathbf{in}\ \ldots : \mathbf{1} \qquad o < o'}{y : !^{o'}\mathbf{1}.\mathbf{end}_!, x' : ?^{o}\mathbf{1}.\mathbf{end}_? \vdash^{q} \mathbf{let}\ ((), x') = \mathbf{recv}\ x'\ \mathbf{in}\ \mathbf{let}\ y = \mathbf{send}\ ((), y)\ \mathbf{in}\ \ldots : \mathbf{1}}$$

Closures suspend communication, so T-Lam stores the priority bounds of the function body on the function type, and T-App restores them. For instance, $\lambda x.\mathbf{send}\ (x, y)$ is assigned the type $A \multimap^{o,o} S$, *i.e.*, a function which, when applied, starts and finishes communicating at priority o.

$$\frac{\dfrac{}{\mathbf{send} : A \times !^{o}A.S \multimap^{\top,o} S} \qquad \dfrac{\dfrac{}{x : A \vdash^{\bot} x : A} \qquad \dfrac{}{x : A, y : !^{o}A.S \vdash^{\bot} y : !^{o}A.S}}{x : A, y : !^{o}A.S \vdash^{\bot} (x, y) : A \times !^{o}A.S}}{x : A, y : !^{o}A.S \vdash^{o} \mathbf{send}\ (x, y) : S}}{y : !^{o}A.S \vdash^{\bot} \lambda x.\mathbf{send}\ (x, y) : A \multimap^{o,o} S}$$

Typing rules for configurations are at the bottom of fig. 2. Configurations are typed by a judgement $\Gamma \vdash^{\phi} \mathcal{C}$ stating that "a configuration $\mathcal{C}$ with flag ϕ is well typed under typing environment Γ". Configuration typing is based on the standard typing for GV. Terms are embedded either as main or as child threads. The priority bound from the term typing is discarded, as configurations contain no further blocking actions. Main threads are allowed to return a value, whereas child threads are required to return the unit value. Sequents are annotated with a flag ϕ, which ensures that there is at most one main thread.

While our configuration typing is based on the standard typing for GV, it differs on two counts: (i) *we require that child threads return the unit value*, as opposed to a terminated channel; and (ii) *we simplify typing for parallel composition*. In order to guarantee deadlock freedom, in GV each parallel composition must split *exactly one* channel of the channel pseudo-type $S^\sharp$ into two endpoints of type S and $\overline{S}$. Consequently, associativity of parallel composition does not preserve typing. In PGV, we guarantee deadlock freedom using priorities, which removes the need for the channel pseudo-type $S^\sharp$, and simplifies typing for parallel composition, while restoring type preservation for the structural congruence.

Subject reduction. Unlike with previous versions of GV, structural congruence, term reduction, and configuration reduction are all type preserving.

We must show that substitution preserves priority constraints. For this, we prove lemma 1, which shows that values have finished all their communication, and that any priorities in the type of the value come from the typing environment.

Lemma 1. *If* $\Gamma \vdash^p V : T$*, then* $p = \bot$*, and* $min_{\mathrm{pr}}(\Gamma) = min_{\mathrm{pr}}(T)$*.*

Lemma 2 (Substitution).
If $\Gamma, x : U' \vdash^p M : T$ *and* $\Theta \vdash^q V : U'$*, then* $\Gamma, \Theta \vdash^p M\{V/x\} : T$*.*

Lemma 3 (Subject Reduction, $\longrightarrow_M$**).**
If $\Gamma \vdash^p M : T$ *and* $M \longrightarrow_M M'$*, then* $\Gamma \vdash^p M' : T$*.*

Lemma 4 (Subject Congruence, $\equiv$**).**
If $\Gamma \vdash^\phi \mathcal{C}$ *and* $\mathcal{C} \equiv \mathcal{C}'$*, then* $\Gamma \vdash^\phi \mathcal{C}'$*.*

Theorem 1 (Subject Reduction, $\longrightarrow_{\mathcal{C}}$**).**
If $\Gamma \vdash^\phi \mathcal{C}$ *and* $\mathcal{C} \longrightarrow_{\mathcal{C}} \mathcal{C}'$*, then* $\Gamma \vdash^\phi \mathcal{C}'$*.*

Progress and deadlock freedom. PGV satisfies progress, as PGV configurations either reduce or are in normal form. However, the normal forms may seem surprising at first, as evaluating a well-typed PGV term does not necessarily produce *just* a value. If a term returns an endpoint, then its normal form contains a thread which is ready to communicate on the dual of that endpoint. This behaviour is not new to PGV. Let us consider an example, adapted from Lindley and Morris [39], in which a term returns an endpoint linked to an echo server. The echo server receives a value and sends it back unchanged. Consider the program which creates a new channel, with endpoints x and x', spawns off an echo server listening on x, and then returns x':

$$\bullet\ \mathbf{let}\ (x, x') = \mathbf{new}\ \mathbf{in}\ \mathbf{spawn}\ (\lambda().\mathbf{echo}_x); x' \qquad \mathbf{echo}_x \triangleq \mathbf{let}\ (y, x) = \mathbf{recv}\ x\ \mathbf{in}\ \mathbf{let}\ x = \mathbf{send}\ (y, x)\ \mathbf{in}\ \mathbf{close}\ x$$

If we reduce the above program, we get $(\nu x x')(\circ\ \mathbf{echo}_x \parallel \bullet\ x')$. Clearly, no more evaluation is possible, even though the configuration contains the thread $\circ\ \mathbf{echo}_x$, which is blocked on x. In corollary 1 we will show that if a term does not return an endpoint, it must produce *only* a value.

Actions are terms which perform communication actions and which synchronise between two threads. *Ready terms* are terms which perform communication actions, either by themselves, *e.g.*, creating a new channel or thread, or with another thread, *e.g.*, sending or receiving. Progress for the term language is standard for GV, and deviates from progress for linear λ-calculus only in that terms may reduce to values or *ready terms*:

Definition 1 (Actions). *A term* acts on *an endpoint* x *if it is* $\mathbf{send}\ (V, x)$*,* $\mathbf{recv}\ x$*,* $\mathbf{close}\ x$*, or* $\mathbf{wait}\ x$*. A term is an* action *if it acts on some endpoint* x*.*

Definition 2 (Ready Terms). *A term* L *is* ready *if it is of the form* $E[M]$*, where* M *is of the form* $\mathbf{new}$*,* $\mathbf{spawn}\ N$*,* $\mathbf{link}\ (x, y)$*, or* M *acts on* x*. In the latter case, we say that* L *is* ready to act on x*.*

Lemma 5 (Progress, $\longrightarrow_M$**).** *If* $\Gamma \vdash^p M : T$ *and* Γ *contains only session types, then: (a)* M *is a value; (b)* $M \longrightarrow_M N$ *for some* N*; or (c)* M *is ready.*

Canonical forms deviate from those for GV, in that we opt to move all ν-binders to the top. The standard GV canonical form, alternating ν-binders and their corresponding parallel compositions, does not work for PGV, since multiple channels may be split across a single parallel composition.

A configuration either reduces, or it is equivalent to configuration in normal form. Crucial to the normal form is that each term M_i is blocked on the corresponding channel x_i, and hence no two terms act on dual endpoints. Furthermore, no term M_i can perform a communication action by itself, since those are excluded by the definition of actions. Finally, as a corollary, we get that well-typed terms which do not return endpoints return *just* a value:

Definition 3 (Canonical Forms). *A configuration $\mathcal{C}$ is in canonical form if it is of the form $(\nu x_1 x_1') \dots (\nu x_n x_n')(\circ\, M_1 \parallel \cdots \parallel \circ\, M_m \parallel \bullet\, N)$ where no term M_i is a value.*

Lemma 6 (Canonical Forms). *If $\Gamma \vdash^{\bullet} \mathcal{C}$, there exists some $\mathcal{D}$ such that $\mathcal{C} \equiv \mathcal{D}$ and $\mathcal{D}$ is in canonical form.*

Definition 4 (Normal Forms). *A configuration $\mathcal{C}$ is in normal form if it is of the form $(\nu x_1 x_1') \dots (\nu x_n x_n')(\circ\, M_1 \parallel \cdots \parallel \circ\, M_m \parallel \bullet\, V)$ where each M_i is ready to act on x_i.*

Theorem 2 (Progress, $\longrightarrow_{\mathcal{C}}$). *If $\varnothing \vdash^{\bullet} \mathcal{C}$ and $\mathcal{C}$ is in canonical form, then either $\mathcal{C} \longrightarrow_{\mathcal{C}} \mathcal{D}$ for some $\mathcal{D}$; or $\mathcal{C} \equiv \mathcal{D}$ for some $\mathcal{D}$ in normal form.*

Proof (Sketch). Our proof follows that of Kobayashi [34, theorem 2]. We apply lemma 5 to each thread. Either we obtain a reduction, or each child thread is ready and the main thread ready or a value. We pick the ready term L with the smallest priority bound. If L contains new, spawn, or a link, we apply E-New, E-Spawn, or E-Link. Otherwise, L must be ready on some x_i. Linearity guarantees there is some thread L' which acts on x_i'. If L' is ready, priority typing guarantees it is ready on x_i', and we apply E-Send or E-Close. If L' is not ready, it must be the main thread returning a value. We move L into the i^{th} position and repeat until we either find a reduction or reach normal form.

Corollary 1. *If $\varnothing \vdash^{\phi} \mathcal{C}$, $\mathcal{C} \not\longrightarrow_{\mathcal{C}}$, and $\mathcal{C}$ contains no endpoints, then $\mathcal{C} \equiv \phi\, V$ for some value V.*

It follows immediately from theorem 2 and corollary 1 that a term which does not return an endpoint will complete all its communication actions, thus satisfying deadlock freedom.

3 Relation to Priority CP

We present a correspondence between Priority GV and an updated version of Priority CP [20, PCP], which is Wadler's CP [53] with priorities. This correspondence connects PGV to (a relaxed variant of) classical linear logic.

3.1 Revisiting Priority CP

Types (A, B) in PCP correspond to linear logic connectives annotated with priorities $o \in \mathbb{N}$. Typing environments, duality, and the priority function $\mathrm{pr}(\cdot)$ are defined as expected.

$$A, B ::= A \otimes^o B \mid A \parr^o B \mid \mathbf{1}^o \mid \bot^o \mid A \oplus^o B \mid A \,\&^o\, B \mid \mathbf{0}^o \mid \top^o$$

Processes (P, Q) in PCP are defined by the following grammar.

$$\begin{aligned} P, Q ::= {} & x \leftrightarrow y \mid (\nu xy)P \mid (P \parallel Q) \mid \mathbf{0} \\ \mid {} & x[y].P \mid x[].P \mid x(y).P \mid x().P \\ \mid {} & x \triangleleft \mathrm{inl}.P \mid x \triangleleft \mathrm{inr}.P \mid x \triangleright \{\mathrm{inl} : P; \mathrm{inr} : Q\} \mid x \triangleright \{\} \end{aligned}$$

Processes are typed by sequents $P \vdash \Gamma$, which correspond to the one-sided sequents in classical linear logic. Differently from PGV, in PCP we do not need to store the greatest priority on the sequent, as, due to the absence of higher-order functions, we cannot compose processes *sequentially*.

PCP decomposes cut into T-Res and T-Par rules—corresponding to cycle and mix rules, respectively—and guarantees deadlock freedom by using priority constraints, *e.g.*, as in T-Send.

$$\text{T-Res}\ \frac{P \vdash \Gamma, x : A, y : A^{\perp}}{(\nu xy)P \vdash \Gamma} \qquad \text{T-Par}\ \frac{P \vdash \Gamma \qquad Q \vdash \Delta}{P \parallel Q \vdash \Gamma, \Delta} \qquad \text{T-Send}\ \frac{P \vdash \Gamma, y : A, x : B \qquad o < \min_{\mathrm{pr}}(\Gamma, A, B)}{x[y].P \vdash \Gamma, x : A \otimes^{o} B}$$

The main change we make to PCP is *removing commuting conversions* and defining an operational semantics based on structural congruence. Commuting conversions are necessary if we want our reduction strategy to correspond *exactly* to cut elimination. However, from the perspective of process calculi, commuting conversions behave strangely: they allow an input/output action to be moved to the top of a process, thus potentially blocking actions which were previously possible. This makes CP, and Dardha and Gay's PCP [20], non-confluent. As Lindley and Morris [39] show, all communications that can be performed *with* the use of commuting conversions, can also be performed *without* them, using structural congruence.

In particular for PCP, commuting conversions break our intuition that an action with lower priority *occurs before* an action with higher priority. To cite Dardha and Gay [20] "*if a prefix on a channel endpoint x with priority o is pulled out at top level, then to preserve priority constraints in the typing rules [..], it is necessary to increase priorities of all actions after the prefix on x*" by $o + 1$. One benefit of removing commuting conversions is that we no longer need to dynamically change the priorities during reduction, which means that the intuition for priorities holds true in our updated version of PCP. Furthermore, we can safely define reduction on untyped processes, which means that type and priority information is erasable!

We prove closed progress for our updated PCP.

Theorem 3 (Progress, $\Longrightarrow$).
If $P \vdash \varnothing$, then either $P = \mathbf{0}$ or there exists a Q such that $P \Longrightarrow Q$.

3.2 Correspondence between PGV and PCP

We illustrate the relation between PCP and PGV by defining a translation from PCP to PGV. The translation on types is defined as follows:

$$\begin{array}{llllll} (\!|A \otimes^{o} B|\!) = !^{o}\overline{(\!|A|\!)}.(\!|B|\!) & (\!|\mathbf{1}^{o}|\!) = \mathbf{end}_{!}^{o} & (\!|A \parr^{o} B|\!) = ?^{o}(\!|A|\!).(\!|B|\!) & (\!|\bot^{o}|\!) = \mathbf{end}_{?}^{o} \\ (\!|A \oplus^{o} B|\!) = (\!|A|\!) \oplus^{o} (\!|B|\!) & (\!|\mathbf{0}^{o}|\!) = \oplus^{o}\{\} & (\!|A \,\&^{o}\, B|\!) = (\!|A|\!) \,\&^{o}\, (\!|B|\!) & (\!|\top^{o}|\!) = \&^{o}\{\} \end{array}$$

There are two separate translations on processes. The main translation, $(\!|\cdot|\!)_M$, translates processes to *terms*:

$$
\begin{aligned}
(\!|x \leftrightarrow y|\!)_M &= \mathbf{link}\ (x, y) \\
(\!|(\nu x y) P|\!)_M &= \mathbf{let}\ (x, y) = \mathbf{new}\ \mathbf{in}\ (\!|P|\!)_M \\
(\!|P \parallel Q|\!)_M &= \mathbf{spawn}\ (\lambda().(\!|P|\!)_M); (\!|Q|\!)_M \\
(\!|\mathbf{0}|\!)_M &= () \\
(\!|x[].P|\!)_M &= \mathbf{close}\ x; (\!|P|\!)_M \\
(\!|x().P|\!)_M &= \mathbf{wait}\ x; (\!|P|\!)_M \\
(\!|x[y].P|\!)_M &= \mathbf{let}\ (y, z) = \mathbf{new}\ \mathbf{in}\ \mathbf{let}\ x = \mathbf{send}\ (z, x)\ \mathbf{in}\ (\!|P|\!)_M \\
(\!|x(y).P|\!)_M &= \mathbf{let}\ (y, x) = \mathbf{recv}\ x\ \mathbf{in}\ (\!|P|\!)_M \\
(\!|x \triangleleft \mathrm{inl}.P|\!)_M &= \mathbf{let}\ x = \mathbf{select}\ \mathbf{inl}\ x\ \mathbf{in}\ (\!|P|\!)_M \\
(\!|x \triangleleft \mathrm{inr}.P|\!)_M &= \mathbf{let}\ x = \mathbf{select}\ \mathbf{inr}\ x\ \mathbf{in}\ (\!|P|\!)_M \\
(\!|x \triangleright \{\mathrm{inl} : P; \mathrm{inr} : Q\}|\!)_M &= \mathbf{offer}\ x\ \{\mathbf{inl}\ x \mapsto (\!|P|\!)_M; \mathbf{inr}\ x \mapsto (\!|Q|\!)_M\} \\
(\!|x \triangleright \{\}|\!)_M &= \mathbf{offer}\ x\ \{\}
\end{aligned}
$$

Unfortunately, the operational correspondence along $(\!|\cdot|\!)_M$ is unsound, as it translates ν-binders and parallel compositions to **new** and **spawn**, which can reduce to their equivalent configuration constructs using E-New and E-Spawn. The same goes for ν-binders which are inserted when translating bound send to unbound send. For instance, the process $x[y].P$ is blocked, but its translation uses **new** and can reduce. To address this issue, we use a second translation, $(\!|\cdot|\!)_c$, which is equivalent to $(\!|\cdot|\!)_M$ followed by reductions using E-New and E-Spawn:

$$
\begin{aligned}
(\!|(\nu x y) P|\!)_c &= (\nu x y)(\!|P|\!)_c \\
(\!|P \parallel Q|\!)_c &= (\!|P|\!)_c \parallel (\!|Q|\!)_c \\
(\!|x[y].P|\!)_c &= (\nu y z)(\circ\ \mathbf{let}\ x = \mathbf{send}\ (z, x)\ \mathbf{in}\ (\!|P|\!)_M) \\
(\!|x \triangleleft \mathrm{inl}.P|\!)_c &= (\nu y z)(\circ\ \mathbf{let}\ x = \mathbf{close}\ (\mathbf{send}\ (\mathbf{inl}\ y, x)); z\ \mathbf{in}\ (\!|P|\!)_M) \\
(\!|x \triangleleft \mathrm{inr}.P|\!)_c &= (\nu y z)(\circ\ \mathbf{let}\ x = \mathbf{close}\ (\mathbf{send}\ (\mathbf{inr}\ y, x)); z\ \mathbf{in}\ (\!|P|\!)_M) \\
(\!|P|\!)_c &= \circ(\!|P|\!)_M, \quad \text{if none of the above apply}
\end{aligned}
$$

Typing environments are translated pointwise, and sequents $P \vdash \Gamma$ are translated as $(\!|\Gamma|\!) \vdash^{\circ} (\!|P|\!)_c$, where $\circ$ indicates a child thread. Translated processes do not have a main thread. The translations $(\!|\cdot|\!)_M$ and $(\!|\cdot|\!)_c$ preserve typing, and the latter induces a sound and complete operational correspondence.

Lemma 7 (Preservation, $(\!|\cdot|\!)_M$). *If $P \vdash \Gamma$, then $(\!|\Gamma|\!) \vdash^{p} (\!|P|\!)_M : \mathbf{1}$.*

Theorem 4 (Preservation, $(\!|\cdot|\!)_c$). *If $P \vdash \Gamma$, then $(\!|\Gamma|\!) \vdash^{\circ} (\!|P|\!)_c$.*

Lemma 8. *For any P, either:*

- $\circ\,(\!|P|\!)_M = (\!|P|\!)_c$*; or*
- $\circ\,(\!|P|\!)_M \longrightarrow_{\mathcal{C}}^{+} (\!|P|\!)_c$*, and for any $\mathcal{C}$, if $\circ\,(\!|P|\!)_M \longrightarrow_{\mathcal{C}} \mathcal{C}$, then $\mathcal{C} \longrightarrow_{\mathcal{C}}^{\star} (\!|P|\!)_c$.*

Theorem 5 (Operational Correspondence, Soundness, $(\!|\cdot|\!)_c$).
If $P \vdash \Gamma$ and $(\!|P|\!)_c \longrightarrow_{\mathcal{C}} \mathcal{C}$, there exists a Q such that $P \Longrightarrow^{+} Q$ and $\mathcal{C} \longrightarrow_{\mathcal{C}}^{\star} (\!|Q|\!)_c$

Theorem 6 (Operational Correspondence, Completeness, $(\!|\cdot|\!)_c$). *If $P \vdash \Gamma$ and $P \Longrightarrow Q$, then $(\!|P|\!)_c \longrightarrow_C^+ (\!|Q|\!)_c$.*

4 Related Work and Discussion

Deadlock freedom and progress. Deadlock freedom and progress are well studied properties in the π-calculus. For the 'standard' typed π-calculus, an important line of work starts from Kobayashi's approach to deadlock freedom [33], where priorities are values from an abstract poset. Kobayashi [34] simplifies the abstract poset to pairs of naturals, called *obligations* and *capabilities*. Padovani simplifies these further to a single natural, called a *priority* [44], and adapts obligations/capabilities to session types [43].

For the session-typed π-calculus, Dezani *et al.* [25] guarantee progress by allowing only one active session at a time. Dezani [24] introduces a partial order on channels, similar to Kobayashi [33]. Carbone and Debois [11] define progress for session typed π-calculus in terms of a *catalyser* which provides the missing counterpart to a process. Carbone *et al.* [10] use catalysers to show that progress is a compositional form of lock-freedom and can be lifted to session types via the encoding of session types to linear types [35,21,18]. Vieira and Vasconcelos [51] use single priorities and an abstract partial order to guarantee deadlock freedom in a binary session-typed π-calculus and building on conservation types.

While our work focuses on *binary* session types, it is worth to discuss related work on Multiparty Session Types (MPST). The line of work on MPST starts with Honda *et al.* [31], which guarantees deadlock freedom *within a single session*, but not for session interleaving. Bettini *et al.* [7] follow a technique similar to Kobayashi's for MPST. The main difference with our work is that we associate priorities with communication actions, where Bettini *et al.* [7] associate them with channels. Carbone and Montesi [13] combine MPST with choreographies and obtain a formalism that satisfies deadlock freedom. Deniélou and Yoshida [23] introduce *multiparty compatibility* which generalises duality in binary session types. They synthesise safe and deadlock-free global types from local types leveraging LTSs and communicating automata. Castellani *et al.* [16] guarantee lock freedom, a stronger property than deadlock freedom, for MPST with *internal delegation*, where participants in the same session are allowed to delegate tasks to each other, and internal delegation is captured by the global type. Scalas and Yoshida [46] provide a revision of the foundations for MPST, and offer a less complicated and more general theory, by removing duality/consistency. The type systems is parametric and type checking is decidable, but allows for a novel integration of model checking techniques. More protocols and processes can be typed and are guaranteed to be free of deadlocks.

Neubauer and Thiemann [42] and Vasconcelos *et al.* [49,50] introduce the first functional language with session types. Such works did not guarantee deadlock freedom until GV [39,53]. Toninho *et al.* [48] present a translation of simply-typed λ-calculus into session-typed π-calculus, but their focus is not on deadlock freedom.

Ties with logic. The correspondence between logic and types lays the foundation for functional programming [54]. Since its inception by Girard [27], linear logic has been a candidate for a foundational correspondence for concurrent programs. A correspondence with linear π-calculus was established early on by Abramsky [1] and Bellin and Scott [5]. Many years later, several correspondences between linear logic and the π-calculus with binary session types were proposed. Caires and Pfenning [9] propose a correspondence with dual intuitionistic linear logic, while Wadler [53] proposes a correspondence with classical linear logic. Both guarantee deadlock freedom as a consequence of cut elimination. Dardha and Gay [20] integrate Kobayashi and Padovani's work on priorities [34,44] with CP, loosening its ties to linear logic in exchange for expressivity. Dardha and Pérez [22] compare priorities à la Kobayashi with tree restrictions à la CP, and show that the latter is a subsystem of the former. Balzer *et al.* [2] introduce sharing at the cost of deadlock freedom, which they restore using an approach similar to priorities [3]. Carbone *et al.* [15,12] give a logical view of MPST with a generalised duality. Caires and Pérez [8] give a presentation of MPST in terms of binary session types and the use of a *medium process* which guarantee protocol fidelity and deadlock freedom. Their binary session types are rooted in linear logic. Ciobanu and Horne [17] give the first Curry-Howard correspondence between MPST and BV [28], a conservative extension of linear logic with a non-commutative operator for sequencing. Horne [32] give a system for subtyping and multiparty compatibility where compatible processes are race free and deadlock free using a Curry-Howard correspondence, similar to the approach in [17].

Conclusion. We answered our research question by presenting Priority GV, a session-typed functional language which allows cyclic communication structures and uses priorities to ensure deadlock freedom. We showed its relation to Priority CP [20] via an operational correspondence.

Future work. Our formalism so far only captures the core of GV. In future work, we plan to explore recursion, following Lindley and Morris [40] and Padovani and Novara [45], and sharing, following Balzer and Pfenning [2] or Voinea *et al.* [52].

Acknowledgements. The authors would like to thank Simon Fowler, April Gonçalves, and Philip Wadler for their comments on the manuscript.

References

1. Abramsky, S.: Proofs as processes. Theor. Comput. Sci. **135**(1), 5–9 (1994). https://doi.org/10.1016/0304-3975(94)00103-0
2. Balzer, S., Pfenning, F.: Manifest sharing with session types. Proc. ACM Program. Lang. **1**(ICFP), 37:1–37:29 (2017). https://doi.org/10.1145/3110281
3. Balzer, S., Toninho, B., Pfenning, F.: Manifest deadlock-freedom for shared session types. In: Proc. of ESOP. Lect. Notes Comput. Sci., vol. 11423, pp. 611–639. Springer (2019). https://doi.org/10.1007/978-3-030-17184-1˙22
4. Barber, A.: Dual intuitionistic linear logic (1996), `https://www.lfcs.inf.ed.ac.uk/reports/96/ECS-LFCS-96-347/ECS-LFCS-96-347.pdf`
5. Bellin, G., Scott, P.J.: On the π-calculus and linear logic. Theor. Comput. Sci. **135**(1), 11–65 (1994). https://doi.org/10.1016/0304-3975(94)00104-9
6. Bernardy, J.P., Boespflug, M., Newton, R.R., Jones, S.P., Spiwack, A.: Linear haskell: practical linearity in a higher-order polymorphic language. Proc. of POPL **2**, 1–29 (2018). https://doi.org/10.1145/3158093
7. Bettini, L., Coppo, M., D'Antoni, L., Luca, M.D., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: Proc. of CONCUR. Lect. Notes Comput. Sci., vol. 5201, pp. 418–433. Springer (2008). https://doi.org/10.1007/978-3-540-85361-9˙33
8. Caires, L., Pérez, J.A.: Multiparty session types within a canonical binary theory, and beyond. In: Proc. of FORTE. Lect. Notes Comput. Sci., vol. 9688, pp. 74–95. Springer (2016). https://doi.org/10.1007/978-3-319-39570-8˙6
9. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Proc. of CONCUR. Lect. Notes Comput. Sci., vol. 6269, pp. 222–236. Springer (2010). https://doi.org/10.1007/978-3-642-15375-4˙16
10. Carbone, M., Dardha, O., Montesi, F.: Progress as compositional lock-freedom. In: Proc. of COORDINATION. Lect. Notes Comput. Sci., vol. 8459, pp. 49–64. Springer (2014). https://doi.org/10.1007/978-3-662-43376-8˙4
11. Carbone, M., Debois, S.: A graphical approach to progress for structured communication in web services. In: Proc. of ICE. Electron. Proc. in Theor. Comput. Sci., vol. 38, pp. 13–27 (2010). https://doi.org/10.4204/EPTCS.38.4
12. Carbone, M., Lindley, S., Montesi, F., Schürmann, C., Wadler, P.: Coherence generalises duality: A logical explanation of multiparty session types. In: Proc. of CONCUR. LIPIcs, vol. 59, pp. 33:1–33:15. Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.33
13. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: Multiparty asynchronous global programming. In: Proc. of POPL. pp. 263–274 (2013). https://doi.org/10.1145/2480359.2429101
14. Carbone, M., Montesi, F., Schürmann, C.: Choreographies, logically. Distrib. Comput. **31**(1), 51–67 (2018). https://doi.org/10.1007/978-3-662-44584-6˙5
15. Carbone, M., Montesi, F., Schürmann, C., Yoshida, N.: Multiparty session types as coherence proofs. In: Proc. of CONCUR. LIPIcs, vol. 42, pp. 412–426. Leibniz-Zentrum für Informatik (2015). https://doi.org/10.1007/s00236-016-0285-y
16. Castellani, I., Dezani-Ciancaglini, M., Giannini, P., Horne, R.: Global types with internal delegation. Theor. Comput. Sci. **807**, 128–153 (2020). https://doi.org/10.1016/j.tcs.2019.09.027
17. Ciobanu, G., Horne, R.: Behavioural analysis of sessions using the calculus of structures. In: Proc. of PSI. Lect. Notes Comput. Sci., vol. 9609, pp. 91–106. Springer (2015). https://doi.org/10.1007/978-3-319-41579-6˙8

18. Dardha, O.: Recursive session types revisited. In: Proc. of BEAT. Electron. Proc. in Theor. Comput. Sci., vol. 162, pp. 27–34 (2014). https://doi.org/10.4204/EPTCS.162.4
19. Dardha, O., Gay, S.J.: A new linear logic for deadlock-free session-typed processes. In: Proc. of FoSSaCS. Lect. Notes Comput. Sci., vol. 10803, pp. 91–109. Springer (2018). https://doi.org/10.1007/978-3-319-89366-2˙5
20. Dardha, O., Gay, S.J.: A new linear logic for deadlock-free session typed processes. In: Proc. of FoSSaCS (2018), `http://www.dcs.gla.ac.uk/~ornela/publications/DG18-Extended.pdf`
21. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. In: Proc. of PPDP. pp. 139–150. ACM (2012). https://doi.org/10.1145/2370776.2370794
22. Dardha, O., Pérez, J.A.: Comparing type systems for deadlock-freedom (2018), `https://arxiv.org/abs/1810.00635`
23. Deniélou, P., Yoshida, N.: Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In: Proc. of ICALP. Lect. Notes Comput. Sci., vol. 7966, pp. 174–186. Springer (2013). https://doi.org/10.1007/978-3-642-39212-2˙18
24. Dezani-Ciancaglini, M., de'Liguoro, U., Yoshida, N.: On progress for structured communications. In: Proc. of TGC. Lect. Notes Comput. Sci., vol. 4912, pp. 257–275. Springer (2007). https://doi.org/10.1007/978-3-540-78663-4˙18
25. Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., Drossopoulou, S.: Session types for object-oriented languages. In: Proc. of ECOOP. Lect. Notes Comput. Sci., vol. 4067, pp. 328–352. Springer (2006). https://doi.org/10.1007/11785477˙20
26. Gay, S.J., Vasconcelos, V.T.: Linear type theory for asynchronous session types. J. Funct. Program. **20**(1), 19–50 (2010). https://doi.org/10.1017/S0956796809990268
27. Girard, J.Y.: Linear logic. Theor. Comput. Sci. **50**, 1–102 (1987). https://doi.org/10.1016/0304-3975(87)90045-4
28. Guglielmi, A.: A system of interaction and structure. ACM Trans. Comput. Log. **8**(1), 1 (2007). https://doi.org/10.1145/1182613.1182614
29. Honda, K.: Types for dyadic interaction. In: Proc. of CONCUR. Lect. Notes Comput. Sci., vol. 715, pp. 509–523. Springer (1993). https://doi.org/10.1007/3-540-57208-2˙35
30. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Proc. of ESOP. Lect. Notes Comput. Sci., vol. 1381, pp. 122–138. Springer (1998). https://doi.org/10.1007/BFb0053567
31. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proc. of POPL. vol. 43(1), pp. 273–284. ACM (2008). https://doi.org/10.1145/2827695
32. Horne, R.: Session subtyping and multiparty compatibility using circular sequents. In: Proc. of CONCUR. LIPIcs, vol. 171, pp. 12:1–12:22. Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.CONCUR.2020.12
33. Kobayashi, N.: A partially deadlock-free typed process calculus. ACM Trans. Program. Lang. Syst. **20**(2), 436–482 (1998). https://doi.org/10.1145/276393.278524
34. Kobayashi, N.: A new type system for deadlock-free processes. In: Proc. of CONCUR. Lect. Notes Comput. Sci., vol. 4137, pp. 233–247. Springer (2006). https://doi.org/10.1007/11817949˙16
35. Kobayashi, N.: Type systems for concurrent programs (2007)
36. Kokke, W., Dardha, O.: Deadlock-free session types in Linear Haskell (2021), `https://arxiv.org/abs/2103.14481`

37. Kokke, W., Montesi, F., Peressotti, M.: Better late than never: A fully-abstract semantics for classical processes. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290337
38. Kokke, W., Montesi, F., Peressotti, M.: Taking linear logic apart. In: Proc. of Linearity & TLLA. Electron. Proc. in Theor. Comput. Sci., vol. 292, pp. 90–103. Open Publishing Association (2019). https://doi.org/10.4204/EPTCS.292.5
39. Lindley, S., Morris, J.G.: A semantics for propositions as sessions. In: Proc. of ESOP. pp. 560–584 (2015). https://doi.org/10.1007/978-3-662-46669-8˙23
40. Lindley, S., Morris, J.G.: Talking bananas: Structural recursion for session types. In: Proc. of ICFP. ACM (2016). https://doi.org/10.1145/2951913.2951921
41. Montesi, F., Peressotti, M.: Classical transitions (2018), `https://arxiv.org/abs/1803.01049`
42. Neubauer, M., Thiemann, P.: An implementation of session types. In: Jayaraman, B. (ed.) Proc. of PADL. Lect. Notes Comput. Sci., vol. 3057, pp. 56–70. Springer (2004). https://doi.org/10.1007/978-3-540-24836-1_5
43. Padovani, L.: From lock freedom to progress using session types. In: Proc. of PLACES. vol. 137, pp. 3–19. Electron. Proc. in Theor. Comput. Sci. (2013). https://doi.org/10.4204/EPTCS.137.2
44. Padovani, L.: Deadlock and lock freedom in the linear π-calculus. In: Proc. of CSL-LICS. pp. 72:1–72:10. ACM (2014). https://doi.org/10.1145/2603088.2603116
45. Padovani, L., Novara, L.: Types for deadlock-free higher-order programs. In: Proc. of FORTE. Lect. Notes Comput. Sci., vol. 9039, pp. 3–18. Springer (2015). https://doi.org/10.1007/978-3-319-19195-9˙1
46. Scalas, A., Yoshida, N.: Less is more: Multiparty session types revisited. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290343
47. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Proc. of PARLE. Lect. Notes Comput. Sci., vol. 817, pp. 398–413. Springer (1994). https://doi.org/10.1007/3-540-58184-7˙118
48. Toninho, B., Caires, L., Pfenning, F.: Functions as session-typed processes. In: Proc. of FoSSaCS. Lect. Notes Comput. Sci., vol. 7213, pp. 346–360. Springer (2012). https://doi.org/10.1007/978-3-642-28729-9˙23
49. Vasconcelos, V., Ravara, A., Gay, S.J.: Session types for functional multithreading. In: Proc. of CONCUR. Lect. Notes Comput. Sci., vol. 3170, pp. 497–511. Springer (2004). https://doi.org/10.1007/978-3-540-28644-8˙32
50. Vasconcelos, V.T., Gay, S.J., Ravara, A.: Type checking a multithreaded functional language with session types. Theor. Comput. Sci. **368**(1-2), 64–87 (2006). https://doi.org/10.1016/j.tcs.2006.06.028
51. Vieira, H.T., Vasconcelos, V.T.: Typing progress in communication-centred systems. In: Proc. of COORDINATION. Lect. Notes Comput. Sci., vol. 7890, pp. 236–250. Springer (2013). https://doi.org/10.1007/978-3-662-43376-8˙10
52. Voinea, A.L., Dardha, O., Gay, S.J.: Resource sharing via capability-based multiparty session types. In: Proc. of IFM. Lect. Notes Comput. Sci., vol. 11918, pp. 437–455. Springer (2019). https://doi.org/10.1007/978-3-030-34968-4˙24
53. Wadler, P.: Propositions as sessions. J. Funct. Program. **24**(2-3), 384–418 (2014). https://doi.org/10.1017/S095679681400001X
54. Wadler, P.: Propositions as types. Commun. ACM **58**(12), 75–84 (2015). https://doi.org/10.1145/2699407