Separating Sessions Smoothly

Simon Fowler ✉
University of Glasgow, UK

Wen Kokke ✉
The University of Edinburgh, UK

Ornela Dardha ✉
University of Glasgow, UK

Sam Lindley ✉
The University of Edinburgh, UK

J. Garrett Morris ✉
The University of Iowa, USA

Abstract

This paper introduces Hypersequent GV (HGV), a modular and extensible core calculus for functional programming with session types that enjoys deadlock freedom, confluence, and strong normalisation. HGV exploits hyper-environments, which are collections of type environments, to ensure that structural congruence is type preserving. As a consequence we obtain a tight operational correspondence between HGV and HCP, a hypersequent-based process-calculus interpretation of classical linear logic. Our translations from HGV to HCP and vice-versa both preserve and reflect reduction. HGV scales smoothly to support Girard's Mix rule, a crucial ingredient for channel forwarding and exceptions.

1 Introduction

Session types [19, 45, 20] are types used to verify communication protocols in concurrent and distributed systems: just as data types rule out dividing an integer by a string, session types rule out sending along an input channel. Session types originated in process calculi, but there is a gap between process calculi, which model the evolving state of concurrent systems, and the descriptions of these systems in typical programming languages. This paper addresses two foundations for session types: (1) a session-typed concurrent lambda calculus called GV [31], intended to be a modular and extensible basis for functional programming languages with session types; and, (2) a session-typed process calculus called CP [51], with a propositions-as-types correspondence to classical linear logic (CLL) [18].

Processes in CP correspond exactly to proofs in CLL and deadlock freedom follows from cut-elimination for CLL. However, while CP is strongly tied to CLL, at the same time it departs from π-calculus. Independent π-calculus features can only appear in combination in CP: CP combines name restriction with parallel composition ($(\nu x)(P \parallel Q)$), corresponding to CLL's cut rule, and combines sending (of bound names only) with parallel composition ($x[y].(P \parallel Q)$), corresponding to CLL's tensor rule. This results in a proliferation of process constructors and prevents the use of standard techniques from concurrency theory, such as labelled-transition semantics and bisimulation. Hypersequent CP (HCP) [34, 28, 27] restores the independence of these features, factoring out parallel composition into a standalone construct while retaining the close correspondence with CLL proofs. HCP typing reasons about collections of processes using collections of type environments (or *hyper-environments*).

GV extends linear λ-calculus with constants for session-typed communication. Following Gay and Vasconcelos [17], Lindley and Morris [31] describe GV's semantics by combining a reduction relation on single terms, following standard λ-calculus rules, and a reduction relation on concurrent configurations of terms, following standard π-calculus rules. They then give a semantic characterisation of deadlocked processes, an extrinsic [42] type system for configurations, and show that well-typed configurations are deadlock-free. There is, however, a large fly in this otherwise smooth ointment: process equivalence does not preserve typing. As a result, it is not enough for Lindley and Morris to show progress and preservation for well-typed configurations; instead, they must show progress and preservation for *all* configurations *equivalent to* well-typed configurations. This not only complicates the metatheory of GV, but the burden is inherited by any effort to build on GV's account of concurrency [15].

In this paper, we show that using hyper-environments in the typing of configurations enables a metatheory for GV that, compared to that of Lindley and Morris, is simpler, is more general, and as a result is easier to use and easier to extend. Hypersequent GV (HGV) repairs the treatment of process equivalence—equivalent configurations are equivalently typeable—and avoids the need for formal gimmickry connecting name restriction and parallel composition. HGV admits standard semantic techniques for concurrent programs: we use bisimulation to show that our translations both preserve *and reflect* reduction, whereas Lindley and Morris show only that their translations between GV and CP preserve reduction as well as resorting to weak explicit substitutions [29]. HGV is also more easily extensible: we outline three examples, including showing that HGV naturally extends to disconnected sets of communication processes, without any change to the proof of deadlock freedom, and that it serves as a simpler foundation for existing work on exceptions in GV [15].

Contributions The paper contributes the following:

- Section 3 introduces Hypersequent GV (HGV), a modular and extensible core calculus for functional programming with session types which uses hyper-environments to ensure that structural congruence is type preserving.
- Section 4 shows that every well-typed GV configuration is also a well-typed HGV configuration, and every tree-structured HGV configuration is equivalent to a well-typed GV configuration.
- Section 5 gives a tight operational correspondences between HGV and HCP via translations in both directions that preserve and reflect reduction.
- Section 6 demonstrates the extensibility of HGV through: (1) unconnected processes, (2) a simplified treatment of forwarding, and (3) an improved foundation for exceptions.

Section 2 reviews GV and its metatheory, Section 7 discusses related work, and Section 8 concludes and discusses future work.

2 The Equivalence Embroglio

GV programs are deadlock free, which GV ensures by restricting process structures to trees. A *process structure* is an undirected graph where nodes represent processes and edges represent channels shared between the connected nodes. Session-typed programs with an acyclic process structure are deadlock-free by construction. We illustrate this with a session-typed vending machine example written in GV.

▶ **Example 2.1.** Consider the session type of a vending machine below, which sells candy bars and lollipops. If the vending machine is free, the customer can press ① to receive a candy bar or ② to receive a lollipop. If the vending machine is busy, the session ends.

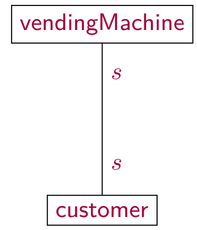
$$\mathsf{VendingMachine} \triangleq \oplus \left\{ \begin{array}{lll} \mathsf{Free} & : & \& \{ ① : !\mathsf{CandyBar}.\mathbf{end}_!, ② : !\mathsf{Lollipop}.\mathbf{end}_! \} \\ \mathsf{Busy} & : & \mathbf{end}_! \end{array} \right\}$$

The customer's session type is *dual*: where the vending machine sends a CandyBar, the customer receives a CandyBar, and so forth. Figure 1 shows the vending machine and customer as a GV program with its process structure.

GV establishes the restriction to tree-structured processes by restricting the primitive for spawning processes. In GV, **fork** has type $(S \multimap \mathbf{end}_!) \multimap \overline{S}$. It takes a closure of type $S \multimap \mathbf{end}_!$ as an argument, creates a channel with endpoints of dual types S and $\overline{S}$, spawns

$$
\begin{array}{l}
\textbf{let } \mathsf{vendingMachine} = \lambda s. \\
\quad \textbf{let } s = \textbf{select } \mathsf{Free}\ s \textbf{ in} \\
\quad \textbf{let } s = \textbf{offer } s \begin{Bmatrix} ① \mapsto \textbf{send } \mathsf{candyBar} \\ ② \mapsto \textbf{send } \mathsf{lollipop} \end{Bmatrix} \\
\quad \textbf{close } s \\
\textbf{in let } \mathsf{customer} = \lambda s. \\
\quad \textbf{offer } s \left\{ \begin{array}{lll} \mathsf{Free} & \mapsto & \textbf{let } s = \textbf{select } ①\ s \textbf{ in} \\ & & \textbf{let } (\mathsf{cb}, s) = \textbf{recv } s \textbf{ in} \\ & & \textbf{wait } s; \textbf{eat } cb \\ \mathsf{Busy} & \mapsto & \textbf{wait } s; \textbf{hungry} \end{array} \right\} \\
\textbf{in let } s = \textbf{fork } (\lambda s. \mathsf{vendingMachine}\ s) \\
\textbf{in } \mathsf{customer}\ s
\end{array}
$$

(a) Vending machine and customer as a GV program.



(b) Process structure of Figure 1a.

Figure 1 Example program with process structure.

the closure as a new process by supplying one of the endpoints as an argument, and then returns the other endpoint. In essence, **fork** is a branching operation on the process structure: it creates a new node connected to the current node by a single edge. Linearity guarantees that the tree structure is preserved, even in the presence of higher-order channels.

Lindley and Morris [31] introduce a semantics for GV, which evaluates programs embedded in process configurations, consisting of embedded programs, flagged as main ($\bullet$) or child ($\circ$) threads, ν-binders to create new channels, and parallel compositions:

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

They introduce these process configurations together with a standard structural congruence, which allows, amongst other things, the reordering of processes using commutativity ($\mathcal{C} \parallel \mathcal{C}' \equiv \mathcal{C}' \parallel \mathcal{C}$), associativity ($\mathcal{C} \parallel (\mathcal{C}' \parallel \mathcal{C}'') \equiv (\mathcal{C} \parallel \mathcal{C}') \parallel \mathcal{C}''$), and scope extrusion ($\mathcal{C} \parallel (\nu x)\mathcal{C}' \equiv (\nu x)(\mathcal{C} \parallel \mathcal{C}')$ if $x \notin \mathrm{fv}(\mathcal{C})$). They guarantee acyclicity by defining an extrinsic type system for configurations. In particular, the type system requires that in every parallel composition $\mathcal{C} \parallel \mathcal{D}$, $\mathcal{C}$ and $\mathcal{D}$ must have exactly one channel in common, and that in a name restriction $(\nu x)\mathcal{C}$, channel x cannot be used until it is shared across a parallel composition.

These restrictions are sufficient to guarantee deadlock freedom. Unfortunately, however, they are not preserved by process equivalence. As Lindley and Morris write:

> Alas, our notion of typing is not preserved by configuration equivalence. For example, assume that $\Gamma \vdash (\nu xy)(C_1 \parallel (C_2 \parallel C_3))$, where $x \in \mathrm{fv}(C_1), y \in \mathrm{fv}(C_2)$, and $x, y \in \mathrm{fv}(C_3)$. We have that $C_1 \parallel (C_2 \parallel C_3) \equiv (C_1 \parallel C_2) \parallel C_3$, but $\Gamma \nvdash (\nu xy)((C_1 \parallel C_2) \parallel C_3)$, as both x and y must be shared between the processes $C_1 \parallel C_2$ and C_3.

As a result, standard notions of progress and preservation are not enough to guarantee deadlock freedom, as reduction sequences could include equivalence steps from well-typed to non-well-typed terms! Instead, they must prove a stronger result:

▶ **Theorem 3** (Lindley and Morris [31]). *If $\Gamma \vdash \mathcal{C}$, $\mathcal{C} \equiv \mathcal{C}'$, and $\mathcal{C}' \longrightarrow \mathcal{D}'$, then there exists $\mathcal{D}$ such that $\mathcal{D} \equiv \mathcal{D}'$ and $\Gamma \vdash \mathcal{D}$.*

This is not a one-time cost: languages based on GV must either also give up on type preservation for structural congruence [15] or admit deadlocks [21, 46].

3 Hypersequent GV

We present Hypersequent GV (HGV), a linear λ-calculus extended with session types and primitives for session-typed communication. HGV shares its syntax and static typing with GV, but uses hyper-environments for runtime typing to simplify and generalise its semantics.

Typing rules for terms $\boxed{\Gamma \vdash M : T}$

$$\frac{}{x : T \vdash x : T}\ \textsc{TM-Var} \qquad \frac{}{\cdot \vdash K : T}\ \textsc{TM-Const} \qquad \frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x.M : T \multimap U}\ \textsc{TM-Lam} \qquad \frac{\Gamma \vdash M : T \multimap U \qquad \Delta \vdash N : T}{\Gamma, \Delta \vdash M\ N : U}\ \textsc{TM-App}$$

$$\frac{}{\cdot \vdash () : \mathbf{1}}\ \textsc{TM-Unit} \qquad \frac{\Gamma \vdash M : \mathbf{1} \qquad \Delta \vdash N : T}{\Gamma, \Delta \vdash \mathbf{let}\ () = M\ \mathbf{in}\ N : T}\ \textsc{TM-LetUnit} \qquad \frac{\Gamma \vdash M : T \qquad \Delta \vdash N : U}{\Gamma, \Delta \vdash (M, N) : T \times U}\ \textsc{TM-Pair}$$

$$\frac{\Gamma \vdash M : T \times T' \qquad \Delta, x : T, y : T' \vdash N : U}{\Gamma, \Delta \vdash \mathbf{let}\ (x, y) = M\ \mathbf{in}\ N : U}\ \textsc{TM-LetPair} \qquad \frac{\Gamma \vdash M : \mathbf{0}}{\Gamma \vdash \mathbf{absurd}\ M : T}\ \textsc{TM-Absurd} \qquad \frac{\Gamma \vdash M : T}{\Gamma \vdash \mathbf{inl}\ M : T + U}\ \textsc{TM-Inl}$$

$$\frac{\Gamma \vdash M : U}{\Gamma \vdash \mathbf{inr}\ M : T + U}\ \textsc{TM-Inr} \qquad \frac{\Gamma \vdash L : T + T' \qquad \Delta, x : T \vdash M : U \qquad \Delta, y : T' \vdash N : U}{\Gamma, \Delta \vdash \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} : U}\ \textsc{TM-CaseSum}$$

Type schemas for communication primitives $\boxed{K : T}$

$$\mathbf{link} : S \times \overline{S} \multimap \mathbf{end}_! \qquad \mathbf{send} : T \times {!T.S} \multimap S \qquad \mathbf{wait} : \mathbf{end}_? \multimap \mathbf{1}$$
$$\mathbf{fork} : (S \multimap \mathbf{end}_!) \multimap \overline{S} \qquad \mathbf{recv} : {?T.S} \multimap T \times S$$

Duality $\boxed{\overline{S}}$

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

Figure 2 HGV, duality and typing rules for terms.

Types, terms, and static typing Types (T, U) comprise a unit type ($\mathbf{1}$), an empty type ($\mathbf{0}$), product types ($T \times U$), sum types ($T + S$), linear function types ($T \multimap U$), and session types (S).

$$T, U ::= \mathbf{1} \mid \mathbf{0} \mid T \times U \mid T + U \mid T \multimap U \mid S \qquad S ::= {!T.S} \mid {?T.S} \mid \mathbf{end}_! \mid \mathbf{end}_?$$

Session types (S) comprise output ($!T.S$: send a value of type T, then behave like S), input ($?T.S$: receive a value of type T, then behave like S), and dual end types ($\mathbf{end}_!$ and $\mathbf{end}_?$). The dual end points restrict process structure to *trees* [51]; conflating them loosens this restriction to *forests* [3]. We let Γ, Δ range over type environments.

The terms and typing rules are given in Figure 2. The linear λ-calculus rules are standard. Each communication primitive has a type schema: **link** takes a pair of compatible endpoints and forwards all messages between them; **fork** takes a function, which is passed one endpoint (of type S) of a fresh channel yielding a new child thread, and returns the other endpoint (of type $\overline{S}$); **send** takes a pair of a value and an endpoint, sends the value over the endpoint, and returns an updated endpoint; **recv** takes an endpoint, receives a value over the endpoint, and returns the pair of the received value and an updated endpoint; and **wait** synchronises on a terminated endpoint of type $\mathbf{end}_?$. Output is dual to input, and $\mathbf{end}_!$ is dual to $\mathbf{end}_?$. Duality is involutive, *i.e.*, $\overline{\overline{S}} = S$.

We write $M; N$ for $\mathbf{let}\ () = M\ \mathbf{in}\ N$, $\mathbf{let}\ x = M\ \mathbf{in}\ N$ for $(\lambda x.N)\ M$, $\lambda().M$ for $\lambda z.z; M$, and $\lambda(x, y).M$ for $\lambda z.\mathbf{let}\ (x, y) = z\ \mathbf{in}\ M$. We write $K : T$ for $\cdot \vdash K : T$ in typing derivations.

▶ Remark 3.1. We include **link** because it is convenient for the correspondence with CP, which interprets CLL's axiom as forwarding. We *can* encode **link** in GV via a type directed translation akin to CLL's *identity expansion*.

Typing rules for configurations $\boxed{\mathcal{G} \vdash \mathcal{C} : R}$

$$\textsc{TC-New}\ \frac{\mathcal{G} \parallel \Gamma, x : S \parallel \Delta, y : \overline{S} \vdash \mathcal{C} : R}{\mathcal{G} \parallel \Gamma, \Delta \vdash (\nu xy)\mathcal{C} : R} \qquad \textsc{TC-Par}\ \frac{\mathcal{G} \vdash \mathcal{C} : R \qquad \mathcal{H} \vdash \mathcal{D} : R'}{\mathcal{G} \parallel \mathcal{H} \vdash \mathcal{C} \parallel \mathcal{D} : R \sqcap R'}$$

$$\textsc{TC-Main}\ \frac{\Gamma \vdash M : T}{\Gamma \vdash \bullet\, M : \bullet\, T} \qquad \textsc{TC-Child}\ \frac{\Gamma \vdash M : \mathbf{end}_!}{\Gamma \vdash \circ\, M : \circ} \qquad \textsc{TC-Link}\ \frac{}{x : S, y : \overline{S}, z : \mathbf{end}_? \vdash x \overset{z}{\leftrightarrow} y : \circ}$$

Configuration types **Configuration type combination** $\boxed{R \sqcap R'}$

$$R ::= \circ \mid \bullet\, T \qquad \bullet\, T \sqcap \circ = \bullet\, T \qquad \circ \sqcap \bullet\, T = \bullet\, T \qquad \circ \sqcap \circ = \circ$$

Figure 3 HGV, typing rules for configurations.

Configurations and runtime typing Process configurations ($\mathcal{C}, \mathcal{D}, \mathcal{E}$) comprise child threads ($\circ\, M$), the main thread ($\bullet\, M$), link threads ($x \overset{z}{\leftrightarrow} y$), name restrictions ($(\nu xy)\mathcal{C}$), and parallel compositions ($\mathcal{C} \parallel \mathcal{D}$). We refer to a configuration of the form $\circ M$ or $x \overset{z}{\leftrightarrow} y$ as an *auxiliary thread*, and a configuration of the form $\bullet M$ as a *main thread*. We let $\mathcal{A}$ range over auxiliary threads and $\mathcal{T}$ range over all threads (auxiliary or main).

$$\phi ::= \bullet \mid \circ \qquad\qquad \mathcal{C}, \mathcal{D}, \mathcal{E} ::= \phi\ M \mid x \overset{z}{\leftrightarrow} y \mid \mathcal{C} \parallel \mathcal{D} \mid (\nu xy)\mathcal{C}$$

The configuration language is reminiscent of π-calculus processes, but has some non-standard features. Name restriction uses double binders [49] in which one name is bound to each endpoint of the channel. Link threads [32] handle forwarding. A link thread $x \overset{z}{\leftrightarrow} y$ waits for the thread connected to z to terminate before forwarding all messages between x and y.

Configuration typing departs from GV [31], exploiting *hypersequents* [4] to recover modularity and extensibility. Inspired by HCP [34, 28, 27], configurations are typed under a *hyper-environment*, a collection of disjoint type environments. We let $\mathcal{G}, \mathcal{H}$ range over hyper-environments, writing $\varnothing$ for the empty hyper-environment, $\mathcal{G} \parallel \Gamma$ for disjoint extension of $\mathcal{G}$ with type environment Γ, and $\mathcal{G} \parallel \mathcal{H}$ for disjoint concatenation of $\mathcal{G}$ and $\mathcal{H}$.

The typing rules for configurations are given in Figure 3. Rules TC-New and TC-Par are key to deadlock freedom: TC-New joins two disjoint configurations with a new channel, and merges their type environments; TC-Par combines two disjoint configurations, and registers their disjointness by separating their type environments in the hyper-environment. Rules TC-Main, TC-Child, and TC-Link type main, child, and link threads, respectively; all three require a singleton hyper-environment. A configuration has type $\circ$ if it has no main thread, and $\bullet\ T$ if it has a main thread of type T. The configuration type combination operator ensures that a well-typed configuration has at most one main thread.

Operational semantics HGV values (U, V, W), evaluation contexts (E), and term reduction rules ($\longrightarrow_{\mathsf{M}}$) define a standard call-by-value, left-to-right evaluation strategy (Appendix A). A closed term either reduces to a value or is blocked on a communication action.

Figure 4 gives the configuration reduction rules. Thread contexts (F) extend evaluation contexts to threads, *i.e.*, $F ::= \phi\ E$. The structural congruence rules are standard apart from SC-LinkComm, which ensures links are undirected, and SC-NewSwap, which swaps names in double binders. The concurrent behaviour of HGV is given by a nondeterministic reduction relation ($\longrightarrow$) on configurations. The first two rules, E-Reify-Fork and E-Reify-Link, create child and link threads, respectively. The next three rules, E-Comm-Link, E-Comm-Send, and E-Comm-Close perform communication actions. The final four rules enable reduction under name restriction and parallel composition, rewriting by structural congruence, and term

Structural congruence $\boxed{\mathcal{C} \equiv \mathcal{D}}$

$$
\begin{array}{llll}
\text{SC-ParAssoc} & \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} & \text{SC-ParComm} & \mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C} \\
\text{SC-NewComm} & (\nu xy)(\nu zw)\mathcal{C} \equiv (\nu zw)(\nu xy)\mathcal{C} & \text{SC-NewSwap} & (\nu xy)\mathcal{C} \equiv (\nu yx)\mathcal{C} \\
\text{SC-ScopeExt} & (\nu xy)(\mathcal{C} \parallel \mathcal{D}) \equiv \mathcal{C} \parallel (\nu xy)\mathcal{D},\ \text{if } x, y \notin \text{fv}(\mathcal{C}) & \text{SC-LinkComm} & x \overset{z}{\leftrightarrow} y \equiv y \overset{z}{\leftrightarrow} x
\end{array}
$$

Configuration reduction $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

$$
\begin{array}{llll}
\text{E-Reify-Fork} & F[\mathbf{fork}\ V] & \longrightarrow & (\nu xx')(F[x] \parallel \circ\,(V\ x')),\ \text{where } x, x' \text{ fresh} \\
\text{E-Reify-Link} & F[\mathbf{link}\ (x, y)] & \longrightarrow & (\nu zz')(x \overset{z}{\leftrightarrow} y \parallel F[z']),\ \text{where } z, z' \text{ fresh}
\end{array}
$$

$$
\begin{array}{llll}
\text{E-Comm-Link} & (\nu zz')(\nu xx')(x \overset{z}{\leftrightarrow} y \parallel \circ\, z' \parallel \phi\ M) & \longrightarrow & \phi\ (M\{y/x'\}) \\
\text{E-Comm-Send} & (\nu xy)(F[\mathbf{send}\ (V, x)] \parallel F'[\mathbf{recv}\ y]) & \longrightarrow & (\nu xy)(F[x] \parallel F'[(V, y)]) \\
\text{E-Comm-Close} & (\nu xy)(\circ\, y \parallel F[\mathbf{wait}\ x]) & \longrightarrow & F[()]
\end{array}
$$

$$
\text{E-Res}\ \frac{\mathcal{C} \longrightarrow \mathcal{C}'}{(\nu xy)\mathcal{C} \longrightarrow (\nu xy)\mathcal{C}'}
\qquad
\text{E-Par}\ \frac{\mathcal{C} \longrightarrow \mathcal{C}'}{\mathcal{C} \parallel \mathcal{D} \longrightarrow \mathcal{C}' \parallel \mathcal{D}}
\qquad
\text{E-Equiv}\ \frac{\mathcal{C} \equiv \mathcal{C}' \quad \mathcal{C}' \longrightarrow \mathcal{D}' \quad \mathcal{D}' \equiv \mathcal{D}}{\mathcal{C} \longrightarrow \mathcal{D}}
\qquad
\text{E-Lift-M}\ \frac{M \longrightarrow_{\mathsf{M}} M'}{F[M] \longrightarrow F[M']}
$$

Figure 4 HGV, configuration reduction.

reduction in threads. Two rules handle links: E-Reify-Link creates a new *link thread* $x \overset{z}{\leftrightarrow} y$ which blocks on z of type $\mathbf{end}_?$, one endpoint of a fresh channel. The other endpoint, z' of type $\mathbf{end}_!$, is placed in the evaluation context of the parent thread. When z' terminates a child thread, E-Comm-Link performs forwarding by substitution.

Choice Internal and external choice are encoded with sum types and session delegation [23, 13]. Prior encodings of choice in GV [31] are asynchronous. To encode synchronous choice we add a dummy synchronisation before exchanging the value of sum type, as follows:

$$
\begin{array}{lll}
S \oplus S' & \triangleq & !\mathbf{1}.!(\overline{S_1} + \overline{S_2}).\mathbf{end}_! \\
S \,\&\, S' & \triangleq & ?\mathbf{1}.?(S_1 + S_2).\mathbf{end}_? \\
\oplus\{\} & \triangleq & !\mathbf{1}.!\mathbf{0}.\mathbf{end}_! \\
\&\{\} & \triangleq & ?\mathbf{1}.?\mathbf{0}.\mathbf{end}_?
\end{array}
\qquad
\begin{array}{l}
\mathbf{select}\ \ell \ \triangleq\ \lambda x. \left(\begin{array}{l} \mathbf{let}\ x = \mathbf{send}\ ((), x)\ \mathbf{in} \\ \mathbf{fork}\ (\lambda y.\mathbf{send}\ (\ell\ y, x)) \end{array} \right) \\
\mathbf{offer}\ L\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} \\
\qquad \triangleq\ \begin{array}{l} \mathbf{let}\ ((), z) = \mathbf{recv}\ L\ \mathbf{in}\ \mathbf{let}\ (w, z) = \mathbf{recv}\ z \\ \mathbf{in}\ \mathbf{wait}\ z; \mathbf{case}\ w\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} \end{array} \\
\mathbf{offer}\ L\ \{\}\ \triangleq\ \begin{array}{l} \mathbf{let}\ ((), c) = \mathbf{recv}\ L\ \mathbf{in}\ \mathbf{let}\ (z, c) = \mathbf{recv}\ c \\ \mathbf{in}\ \mathbf{wait}\ c; \mathbf{absurd}\ \ z \end{array}
\end{array}
$$

HGV enjoys type preservation, deadlock freedom, confluence, and strong normalisation (details in Appendix C). Here we outline where the metatheory diverges from GV.

Preservation Hyper-environments enable type preservation under structural congruence, which significantly simplifies the metatheory compared to GV.

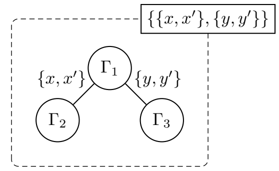
▶ **Theorem 3.2** (Preservation).

1. *If* $\mathcal{G} \vdash \mathcal{C} : R$ *and* $\mathcal{C} \equiv \mathcal{D}$*, then* $\mathcal{G} \vdash \mathcal{D} : R$*.*
2. *If* $\mathcal{G} \vdash \mathcal{C} : R$ *and* $\mathcal{C} \longrightarrow \mathcal{D}$*, then* $\mathcal{G} \vdash \mathcal{D} : R$*.*

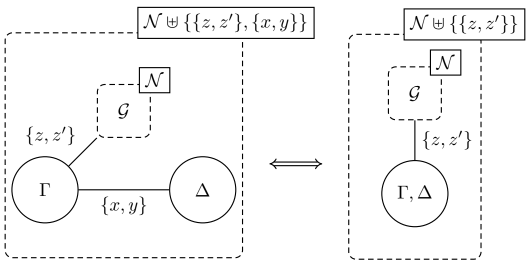
Abstract process structures Unlike in GV, in HGV we cannot rely on the fact that exactly one channel is split over each parallel composition. Instead, we introduce the notion of an *abstract process structure* (APS). An APS is a graph defined over a hyper-environment $\mathcal{G}$ and a set of undirected pairs of co-names (a *co-name set*) $\mathcal{N}$ drawn from the names in $\mathcal{G}$. The nodes of an APS are the type environments in $\mathcal{G}$. Each edge is labelled by a distinct co-name pair $\{x_1, x_2\} \in \mathcal{N}$, such that $x_1 : S \in \Gamma_1$ and $x_2 : \overline{S} \in \Gamma_2$.

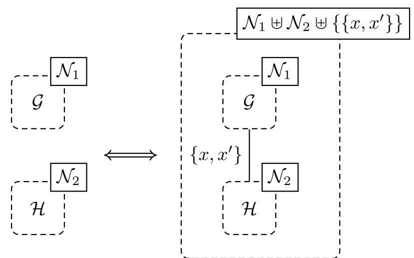
▶ **Example 3.3.**

Let $\mathcal{G} = \Gamma_1 \parallel \Gamma_2 \parallel \Gamma_3$, where $\Gamma_1 = x : S_1, y : S_2$, $\Gamma_2 = x' : \overline{S_1}, z : T$, and $\Gamma_3 = y' : \overline{S_2}$, and suppose $\mathcal{N} = \{\{x, x'\}, \{y, y'\}\}$. The APS for $\mathcal{G}$ and $\mathcal{N}$ is illustrated to the right.



A key feature of HGV is a subformula principle, which states that all hyper-environments arising in the derivation of an HGV program are tree-structured. We write $\mathsf{Tree}(\mathcal{G}, \mathcal{N})$ to denote that the APS for $\mathcal{G}$ with respect to $\mathcal{N}$ is tree-structured. An HGV program $\bullet\, M$ has a single type environment, so is tree-structured; the same goes for child and link threads. Read bottom-up TC-New and TC-Par preserve tree structure: these two properties follow from Lemma B.8 (Appendix B), which is illustrated by the following two pictures.





Tree canonical form We now define a canonical form for configurations that captures the tree structure of an APS. Tree canonical form enables a succinct statement of *open progress* (Lemma 3.8) and a means for embedding HGV in GV (Lemma 4.6).

▶ **Definition 3.4** (Tree canonical form)**.** *A configuration $\mathcal{C}$ is in* tree canonical form *if it can be written:* $(\nu x_1 y_1)(\mathcal{A}_1 \parallel \cdots \parallel (\nu x_n y_n)(\mathcal{A}_n \parallel \phi N) \cdots)$ *where* $x_i \in \mathrm{fv}(\mathcal{A}_i)$ *for* $1 \le i \le n$.

▶ **Theorem 3.5** (Tree canonical form)**.** *If $\Gamma \vdash \mathcal{C} : R$, then there exists some $\mathcal{D}$ such that $\mathcal{C} \equiv \mathcal{D}$ and $\mathcal{D}$ is in tree canonical form.*

▶ **Lemma 3.6.** *If $\Gamma_1 \parallel \cdots \parallel \Gamma_n \vdash \mathcal{C} : R$, then there exist $R_1, \ldots, R_n$ and $\mathcal{D}_1, \ldots, \mathcal{D}_n$ such that $R = R_1 \sqcap \cdots \sqcap R_n$ and $\mathcal{C} \equiv \mathcal{D}_1 \parallel \cdots \parallel \mathcal{D}_n$ and $\Gamma_i \vdash \mathcal{D}_i : R_i$ for each i.*

It follows from Theorem 3.5 and Lemma 3.6 that any well-typed HGV configuration can be written as a forest of independent configurations in tree canonical form.

Progress and Deadlock Freedom

▶ **Definition 3.7** (Blocked thread)**.** *We say that thread $\mathcal{T}$ is* blocked on variable *z, written* $\mathsf{blocked}(\mathcal{T}, z)$, *if either: $\mathcal{T} = \circ\, z$; $\mathcal{T} = x \overset{z}{\leftrightarrow} y$, for some x, y; or $\mathcal{T} = F[N]$ for some F, where N is* **send** (V, z), **recv** z, *or* **wait** z.

We let Ψ range over type environments containing only session-typed variables, *i.e.*, $\Psi ::= \cdot \mid \Psi, x : S$, which lets us reason about configurations that are closed except for runtime names. Using Lemma 3.6 we obtain *open progress* for configurations with free runtime names.

▶ **Lemma 3.8** (Open Progress)**.** *Suppose $\Psi \vdash \mathcal{C} : T$ where $\mathcal{C} = (\nu x_1 y_1)(\mathcal{A}_1 \parallel \cdots \parallel (\nu x_n y_n)(\mathcal{A}_n \parallel \phi N) \cdots)$ is in tree canonical form. Either $\mathcal{C} \longrightarrow \mathcal{D}$ for some $\mathcal{D}$, or:*

1. *For each $\mathcal{A}_j$ $(1 \le j \le n)$, $\mathsf{blocked}(\mathcal{A}_j, z)$ for some $z \in \{x_j\} \cup \{y_k \mid 1 \le k < j\} \cup \mathrm{fv}(\Gamma_i)$*
2. *Either N is a value or $\mathsf{blocked}(\phi N, z)$ for some $z \in \{y_j \mid 1 \le j \le n\} \cup \mathrm{fv}(\Gamma_i)$*

For closed configurations, we obtain a tighter result. If a closed configuration cannot reduce, then each auxiliary thread must either be a value, or be blocked on its neighbouring endpoint.

Typing rules for configurations $\boxed{\Gamma \vdash_{\mathsf{GV}} \mathcal{C} : T}$

$$\text{TG-New}\ \frac{\Gamma, \langle x, y\rangle : S^\sharp \vdash_{\mathsf{GV}} \mathcal{C} : R}{\Gamma \vdash_{\mathsf{GV}} (\nu x y)\mathcal{C} : R} \qquad \text{TG-Connect}_1\ \frac{\Gamma_1, x : S \vdash_{\mathsf{GV}} \mathcal{C} : R \quad \Gamma_2, y : \overline{S} \vdash_{\mathsf{GV}} \mathcal{D} : R'}{\Gamma_1, \Gamma_2, \langle x, y\rangle : S^\sharp \vdash_{\mathsf{GV}} \mathcal{C} \parallel \mathcal{D} : R \sqcap R'} \qquad \text{TG-Connect}_2\ \frac{\Gamma_1, y : \overline{S} \vdash_{\mathsf{GV}} \mathcal{C} : R \quad \Gamma_2, x : S \vdash_{\mathsf{GV}} \mathcal{D} : R'}{\Gamma_1, \Gamma_2, \langle x, y\rangle : S^\sharp \vdash_{\mathsf{GV}} \mathcal{C} \parallel \mathcal{D} : R \sqcap R'}$$

$$\text{TG-Child}\ \frac{\Gamma \vdash_{\mathsf{GV}} M : \mathbf{end}_!}{\Gamma \vdash_{\mathsf{GV}} \circ M : \circ} \qquad \text{TG-Main}\ \frac{\Gamma \vdash_{\mathsf{GV}} M : T}{\Gamma \vdash_{\mathsf{GV}} \bullet M : \bullet\, T} \qquad \text{TG-Link}\ \frac{}{x : S, y : \overline{S}, z : \mathbf{end}_? \vdash_{\mathsf{GV}} x \overset{z}{\leftrightarrow} y : \circ}$$

Figure 5 GV, typing rules for configurations.

Finally, for *ground configurations*, where the main thread does not return a runtime name or capture a runtime name in a closure, we obtain a yet tighter result, *global progress*, which implies deadlock freedom [9].

▶ **Definition 3.9** (Ground configuration). *A configuration $\mathcal{C}$ is a* ground configuration *if $\cdot \vdash \mathcal{C} : T$, $\mathcal{C}$ is in canonical form, and T does not contain session types or function types.*

▶ **Theorem 3.10** (Global progress). *Suppose $\mathcal{C}$ is a ground configuration. Either there exists some $\mathcal{D}$ such that $\mathcal{C} \longrightarrow \mathcal{D}$, or $\mathcal{C} = \bullet V$ for some value V.*

4 Relation between HGV and GV

In this section, we show that well-typed GV configurations are well-typed HGV configurations, and well-typed HGV configurations with tree structure are well-typed GV configuration.

GV HGV and GV share a common term language and reduction semantics, so only differ in their runtime typing rules. Figure 5 gives the runtime typing rules for GV. We adapt the rules to use a double-binder formulation to concentrate on the essence of the relationship with HGV, but it is trivial to translate GV with single binders into GV with double binders.

We require a pseudo-type $S^\sharp$, which is the type of un-split channels and cannot appear in terms. Rule TG-New types a name restriction $(\nu x y)\mathcal{C}$, adding $\langle x, y\rangle : S^\sharp$ to the type environment, which along with TG-Connect$_1$ and TG-Connect$_2$ ensures that a session channel of type S will be split into endpoints x and y over a parallel composition, in turn enforcing a tree process structure. The remaining typing rules are as in HGV.

Embedding GV into HGV Every well-typed open GV configuration is also a well-typed HGV configuration.

▶ **Definition 4.1** (Flattening). *Flattening, written $\downarrow$, converts GV type environments and HGV hyper-environments into HGV environments.*

$$\begin{array}{lcl@{\qquad\qquad}lcl} \downarrow \cdot & = & \cdot & \downarrow \varnothing & = & \varnothing \\ \downarrow(\Gamma, \langle x, x'\rangle : S^\sharp) & = & \downarrow\Gamma, x : S, x' : \overline{S} & \downarrow(\mathcal{G} \parallel \Gamma) & = & \downarrow\mathcal{G}, \Gamma \\ \downarrow(\Gamma, x : T) & = & \downarrow\Gamma, x : T & & & \end{array}$$

▶ **Definition 4.2** (Splitting). *Splitting converts GV typing environments into hyper-environments. Given channels $\{\langle x_i, x_i'\rangle : S_i^\sharp\}_{i \in 1..n}$ in Γ, a hyper-environment $\mathcal{G}$ is a* splitting *of Γ if $\downarrow\mathcal{G} = \downarrow\Gamma$ and $\exists \Gamma_1, \ldots, \Gamma_{n+1}$ such that $\mathcal{G} = \Gamma_1 \parallel \cdots \parallel \Gamma_{n+1}$, and $\mathsf{Tree}(\mathcal{G}, \{\{x_1, x_1'\}, \ldots, \{x_n, x_n'\}\})$.*

A well-typed GV configuration is typeable in HGV under a splitting of its type environment.

▶ **Theorem 4.3** (Typeability of GV configurations in HGV). *If* $\Gamma \vdash_{GV} \mathcal{C} : R$*, then there exists some* $\mathcal{G}$ *such that* $\mathcal{G}$ *is a splitting of* Γ *and* $\mathcal{G} \vdash \mathcal{C} : R$*.*

▶ **Example 4.4.** Consider a configuration where a child thread pings the main thread:

$$(\nu xy)(\circ\,(\mathbf{send}\;(ping, x)) \parallel \bullet\,(\mathbf{let}\;((), y) = \mathbf{recv}\;y\;\mathbf{in}\;\mathbf{wait}\;y))$$

We can write a GV typing derivation as follows:

$$\dfrac{\dfrac{x : !\mathbf{1}.\mathbf{end}_!, ping : \mathbf{1} \vdash_{GV} \circ\,(\mathbf{send}\;(ping, x)) : \circ \qquad y : ?\mathbf{1}.\mathbf{end}_? \vdash_{GV} \bullet\,(\mathbf{let}\;((), y) = \mathbf{recv}\;y\;\mathbf{in}\;\mathbf{wait}\;y) : \bullet\,\mathbf{1}}{\langle x, y\rangle : !\mathbf{1}.\mathbf{end}_!^{\sharp}, ping : \mathbf{1} \vdash_{GV} (\nu xy)(\circ(\mathbf{send}\;(ping, x)) \parallel \bullet(\mathbf{let}\;((), y) = \mathbf{recv}\;y\;\mathbf{in}\;\mathbf{wait}\;y)) : \mathbf{1}}}{ping : \mathbf{1} \vdash_{GV} (\nu xy)(\circ(\mathbf{send}\;(ping, x)) \parallel \bullet(\mathbf{let}\;((), y) = \mathbf{recv}\;y\;\mathbf{in}\;\mathbf{wait}\;y)) : \mathbf{1}}$$

The corresponding HGV derivation is:

$$\dfrac{\dfrac{x : !\mathbf{1}.\mathbf{end}_!, ping : \mathbf{1} \vdash \circ\,(\mathbf{send}\;(ping, x)) : \circ \qquad y : ?\mathbf{1}.\mathbf{end}_? \vdash \bullet\,(\mathbf{let}\;((), y) = \mathbf{recv}\;y\;\mathbf{in}\;\mathbf{wait}\;y) : \bullet\,\mathbf{1}}{x : !\mathbf{1}.\mathbf{end}_!, ping : \mathbf{1} \parallel y : ?\mathbf{1}.\mathbf{end}_? \vdash (\nu xy)(\circ(\mathbf{send}\;(ping, x)) \parallel \bullet(\mathbf{let}\;((), y) = \mathbf{recv}\;y\;\mathbf{in}\;\mathbf{wait}\;y)) : \bullet\,\mathbf{1}}}{ping : \mathbf{1} \vdash (\nu xy)(\circ(\mathbf{send}\;(ping, x)) \parallel \bullet(\mathbf{let}\;((), y) = \mathbf{recv}\;y\;\mathbf{in}\;\mathbf{wait}\;y)) : \bullet\,\mathbf{1}}$$

Note that $x : !\mathbf{1}.\mathbf{end}_!, ping : \mathbf{1} \parallel y : ?\mathbf{1}.\mathbf{end}_?$ is a splitting of $\langle x, y\rangle : (!\mathbf{1}.\mathbf{end}_!)^{\sharp}, ping : \mathbf{1}$.

Translating HGV to GV As we saw earlier, unlike in HGV, equivalence in GV is not type-preserving. It follows that HGV types strictly more processes than GV.

▶ **Theorem 4.5.** *There exist configurations* $\mathcal{C}$ *where* $\cdot \vdash \mathcal{C} : R$ *but* $\cdot \nvdash_{GV} \mathcal{C} : R$*.*

Nevertheless, every well-typed HGV configuration typeable under a singleton hyper-environment Γ is *equivalent* to a well-typed GV configuration, which we show using tree canonical forms.

▶ **Lemma 4.6.** *Suppose* $\Gamma \vdash \mathcal{C} : R$ *where* $\mathcal{C}$ *is in tree canonical form. Then,* $\Gamma \vdash_{GV} \mathcal{C} : R$*.*

▶ Remark 4.7. It is not the case that every HGV configuration typeable under an *arbitrary* hyper-environment $\mathcal{H}$ is equivalent to a well-typed GV configuration. This is because open HGV configurations can form *forest* process structures, whereas (even open) GV configurations must form a *tree* process structure.

Since we can write all well-typed HGV configurations in canonical form, and HGV tree canonical forms are typeable in GV, it follows that every well-typed HGV configuration typeable under a single type environment is equivalent to a well-typed GV configuration.

▶ **Corollary 4.8.** *If* $\Gamma \vdash \mathcal{C} : R$*, then there exists some* $\mathcal{D}$ *such that* $\mathcal{C} \equiv \mathcal{D}$ *and* $\Gamma \vdash_{GV} \mathcal{D} : R$*.*

5 Relation between HGV and HCP

In this section, we explore two translations, from HGV to HCP (in Section 5) and from HCP to HGV (in Section 5), together with their operational correspondences.

Hypersequent CP HCP [34, 28] is a session-typed process calculus with a correspondence to CLL, which exploits hypersequents to fix extensibility and modularity issues with CP.

Types (A, B) consist of the connectives of linear logic: the multiplicative operators ($\otimes$, $⅋$) and units ($\mathbf{1}$, $\bot$) and the additive operators ($\oplus$, $\&$) and units ($\mathbf{0}$, $\top$).

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

Type environments (Γ, Δ) associate names with types. Hyper-environments ($\mathcal{G}$, $\mathcal{H}$) are collections of type environments. The empty type environment and hyper-environment are written $\cdot$ and $\varnothing$, respectively. Names in type and hyper-environments must be unique and environments may be combined, written Γ, Δ and $\mathcal{G} \parallel \mathcal{H}$, only if they are disjoint.

Typing rules for processes $\boxed{P \vdash \mathcal{G}}$

$$\textsc{TP-Link}\ \frac{}{x \leftrightarrow^{A} y \vdash x : A, y : A^{\perp}} \qquad \textsc{TP-New}\ \frac{P \vdash \mathcal{G} \parallel \Gamma, x : A \parallel \Delta, y : A^{\perp}}{(\nu xy)P \vdash \mathcal{G} \parallel \Gamma, \Delta} \qquad \textsc{TP-Par}\ \frac{P \vdash \mathcal{G} \qquad Q \vdash \mathcal{H}}{P \parallel Q \vdash \mathcal{G} \parallel \mathcal{H}} \qquad \textsc{TP-Halt}\ \frac{}{\mathbf{0} \vdash \varnothing} \qquad \textsc{TP-Close}\ \frac{P \vdash \varnothing}{x[].P \vdash x : \mathbf{1}}$$

$$\textsc{TP-Wait}\ \frac{P \vdash \Gamma}{x().P \vdash \Gamma, x : \perp} \qquad \textsc{TP-Send}\ \frac{P \vdash \Gamma, y : A \parallel \Delta, x : B}{x[y].P \vdash \Gamma, \Delta, x : A \otimes B} \qquad \textsc{TP-Recv}\ \frac{P \vdash \Gamma, y : A, x : B}{x(y).P \vdash \Gamma, x : A \parr B} \qquad \textsc{TP-Offer-Absurd}\ \frac{}{x \triangleright \{\} \vdash \Gamma, x : \top}$$

$$\textsc{TP-Select-Inl}\ \frac{P \vdash \Gamma, x : A}{x \triangleleft \mathrm{inl}.P \vdash \Gamma, x : A \oplus B} \qquad \textsc{TP-Select-Inr}\ \frac{P \vdash \Gamma, x : B}{x \triangleleft \mathrm{inr}.P \vdash \Gamma, x : A \oplus B} \qquad \textsc{TP-Offer}\ \frac{P \vdash \Gamma, x : A \qquad Q \vdash \Gamma, x : B}{x \triangleright \{\mathrm{inl} : P; \mathrm{inr} : Q\} \vdash \Gamma, x : A \,\&\, B}$$

Duality $\boxed{A^{\perp}}$

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

Figure 6 HCP, duality and typing rules for processes.

Processes (P, Q) are a variant of the π-calculus with forwarding [44, 7], bound output [44], and double binders [49]. The syntax of processes is given by the typing rules (Figure 6), which are standard for HCP [34, 28]: $x \leftrightarrow y$ forwards messages between x and y; $(\nu xy)P$ creates a channel with endpoints x and y, and continues as P; $P \parallel Q$ composes P and Q in parallel; $\mathbf{0}$ is the terminated process; $x[y].P$ creates a new channel, outputs one endpoint over x, binds the other to y, and continues as P; $x(y).P$ receives a channel endpoint, binds it to y, and continues as P; $x[].P$ and $x().P$ close x and continue as P; $x \triangleleft \mathrm{inl}.P$ and $x \triangleleft \mathrm{inr}.P$ make a binary choice; $x \triangleright \{\mathrm{inl} : P; \mathrm{inr} : Q\}$ offers a binary choice; and $x \triangleright \{\}$ offers a nullary choice. As HCP is synchronous, the only difference between $x[y].P$ and $x(y).P$ is their typing (and similarly for $x[].P$ and $x().P$). We write *unbound* send as $x\langle y\rangle.P$ (short for $x[z].(y \leftrightarrow z \parallel P)$), and synchronisation as $\bar{x}.P$ (short for $x[z].(z[].\mathbf{0} \parallel P)$) and $x.P$ (short for $x(z).z().P$). Duality is standard and is involutive, *i.e.*, $(A^{\perp})^{\perp} = A$.

We define a standard structural congruence ($\equiv$) similar to that of HGV, *i.e.*, parallel composition is commutative and associative, we can commute name restrictions, swap the order of endpoints, swap links, and have scope extrusion (similar to Figure 4).

We define the labeled transition system for HCP as a subsystem of that of Kokke *et al.* [27], omitting delayed actions. Labels ℓ represent the actions a process can take. Prefixes π are a convenient subset which can be written as prefixes to processes, *i.e.*, $\pi.P$. The label τ represents internal actions. We distinguish two subtypes of internal actions: α represents only the evaluation of links as *renaming*, and β represents only *communication*.

$$\begin{array}{lll}
\pi & ::= & x[y] \mid x[] \mid x(y) \mid x() \mid x \triangleleft \mathrm{inl} \mid x \triangleleft \mathrm{inr} \\
\ell & := & \pi \mid x \leftrightarrow y \mid x \triangleright \mathrm{inl} \mid x \triangleright \mathrm{inr} \mid \tau \mid \alpha \mid \beta
\end{array}$$

We let ℓ_x range over labels on x: $x \leftrightarrow y$, $x[y]$, $x[]$, *etc.* Labeled transition $\xrightarrow{\ell}$ is defined in Figure 7. We write $\xrightarrow{\ell}\xrightarrow{\ell'}$ for the composition of $\xrightarrow{\ell}$ and $\xrightarrow{\ell'}$, $\xrightarrow{\ell}_{+}$ for the transitive closure of $\xrightarrow{\ell}$, and $\xrightarrow{\ell}_{\star}$ for the reflexive-transitive closure. We write $\mathrm{bn}(\ell)$ and $\mathrm{fn}(\ell)$ for the bound and free names contained in ℓ, respectively.

The behavioural theory for HCP follows Kokke *et al.* [27], except that we distinguish two subrelations to bisimilarity, following the subtypes of internal actions.

Action rules

$$\textsc{Act-Pref}\quad \pi.P \xrightarrow{\pi} P \qquad \textsc{Act-Link}_1\quad x{\leftrightarrow}y \xrightarrow{x\leftrightarrow y} \mathbf{0} \qquad \textsc{Act-Link}_2\quad x{\leftrightarrow}y \xrightarrow{y\leftrightarrow x} \mathbf{0}$$

$$\textsc{Act-Off-Inl}\quad x \triangleright \{\mathrm{inl} : P; \mathrm{inr} : Q\} \xrightarrow{x\triangleright \mathrm{inl}} P \qquad \textsc{Act-Off-Inr}\quad x \triangleright \{\mathrm{inl} : P; \mathrm{inr} : Q\} \xrightarrow{x\triangleright \mathrm{inr}} Q$$

Communication Rules

$$\textsc{Tau-Alp}\ \frac{P \xrightarrow{\alpha} P'}{P \xrightarrow{\tau} P'} \qquad \textsc{Tau-Bet}\ \frac{P \xrightarrow{\beta} P'}{P \xrightarrow{\tau} P'} \qquad \textsc{Alp-Link}\ \frac{P \xrightarrow{x\leftrightarrow z} P'}{(\nu xy)P \xrightarrow{\alpha} P'\{z/y\}} \qquad \textsc{Bet-Send}\ \frac{P \xrightarrow{x[x'] \parallel y(y')} P'}{(\nu xy)P \xrightarrow{\beta} (\nu xy)(\nu x'y')P'}$$

$$\textsc{Bet-Close}\ \frac{P \xrightarrow{x[] \parallel y()} P'}{(\nu xy)P \xrightarrow{\beta} P'} \qquad \textsc{Bet-Inl}\ \frac{P \xrightarrow{x\triangleleft \mathrm{inl} \parallel y \triangleright \mathrm{inl}} P'}{(\nu xy)P \xrightarrow{\beta} (\nu xy)P'} \qquad \textsc{Bet-Inr}\ \frac{P \xrightarrow{x\triangleleft \mathrm{inr} \parallel y \triangleright \mathrm{inr}} P'}{(\nu xy)P \xrightarrow{\beta} (\nu xy)P'}$$

Structural Rules

$$\textsc{Str-Res}\ \frac{P \xrightarrow{\ell} P' \qquad x, y \notin \mathrm{fn}(\ell)}{(\nu xy)P \xrightarrow{\ell} (\nu xy)P'} \qquad \textsc{Str-Par}_1\ \frac{P \xrightarrow{\ell} P' \qquad \mathrm{bn}(\ell) \cap \mathrm{fn}(Q) = \varnothing}{P \parallel Q \xrightarrow{\ell} P' \parallel Q}$$

$$\textsc{Str-Par}_2\ \frac{Q \xrightarrow{\ell} Q' \qquad \mathrm{bn}(\ell) \cap \mathrm{fn}(P) = \varnothing}{P \parallel Q \xrightarrow{\ell} P \parallel Q'} \qquad \textsc{Str-Syn}\ \frac{P \xrightarrow{\ell} P' \qquad Q \xrightarrow{\ell'} Q' \qquad \mathrm{bn}(\ell) \cap \mathrm{bn}(\ell') = \varnothing}{P \parallel Q \xrightarrow{\ell \parallel \ell'} P' \parallel Q'}$$

Figure 7 HCP, label transition semantics.

▶ **Definition 5.1** (Strong bisimilarity). *A symmetric relation* $\mathcal{R}$ *on processes is a* strong bisimulation *if* $P \mathrel{\mathcal{R}} Q$ *implies that if* $P \xrightarrow{\ell} P'$, *then* $Q \xrightarrow{\ell} Q'$ *for some* Q' *such that* $P' \mathrel{\mathcal{R}} Q'$. Strong bisimilarity *is the largest relation* $\sim$ *that is a strong bisimulation.*

▶ **Definition 5.2** (Saturated transition). *The* ℓ-saturated transition relation, for $\ell \in \{\alpha, \beta, \tau\}$, is the smallest relation $\Longrightarrow_\ell$ such that: $P \overset{\ell}{\Longrightarrow}_\ell P$ for all P; and if $P \overset{\ell}{\Longrightarrow}_\ell P'$, $P' \xrightarrow{\ell'} Q'$, and $Q' \overset{\ell}{\Longrightarrow}_\ell Q$, then $P \overset{\ell'}{\Longrightarrow}_\ell Q$. *Saturated transition*, with no qualifier, refers to the τ-saturated transition relation, and is written $\Longrightarrow$.

▶ **Definition 5.3** (Bisimilarity). *A symmetric relation* $\mathcal{R}$ *on processes is an* ℓ*-bisimulation, for* $\ell \in \{\alpha, \beta, \tau\}$*, if* $P \mathrel{\mathcal{R}} Q$ *implies that if* $P \overset{\ell'}{\Longrightarrow}_\ell P'$*, then* $Q \overset{\ell'}{\Longrightarrow}_\ell Q'$ *for some* Q' *such that* $P' \mathrel{\mathcal{R}} Q'$*. The* ℓ*-bisimilarity relation is the largest relation* $\approx_\ell$ *that is an* ℓ*-bisimulation. Bisimilarity, with no qualifier, refers to* τ*-bisimilarity, and is written* $\approx$*.*

▶ **Lemma 5.4.** *Structural congruence, strong bisimilarity and the various forms of (weak) bisimilarity are in the expected relation,* i.e., $\equiv \subsetneq \sim$, $\sim \subsetneq \approx, \approx_\alpha, \approx_\beta$. *Furthermore, bisimilarity is the union of* α*-bisimilarity and* β*-bisimilarity,* i.e., $\approx = \approx_\alpha \cup \approx_\beta$.

Translating HGV to HCP We factor the translation from HGV to HCP into two translations: (1) a translation into HGV∗, a fine-grain call-by-value [30] variant of HGV, which makes control flow explicit; and (2) a translation from HGV∗ to HCP.

HGV∗ We define HGV∗ as a refinement of HGV in which any non-trivial term must be named by a let binding before being used. While let is syntactic sugar in HGV, it is part of the core language in HGV∗. Correspondingly, the reduction rule for let follows from the

Translation on types $\boxed{\llfloor T \rrfloor \text{ and } \llceil T \rrceil}$

$$\llfloor !T.S \rrfloor = \llfloor T \rrfloor^{\perp} \otimes \llfloor S \rrfloor \qquad \llfloor \mathbf{end}_! \rrfloor = \mathbf{1} \qquad \llfloor T \rrfloor = \llceil T \rrceil^{\perp},$$
$$\llfloor ?T.S \rrfloor = \llfloor T \rrfloor^{\perp} \parr \llfloor S \rrfloor \qquad \llfloor \mathbf{end}_? \rrfloor = \perp \qquad \text{if } T \text{ is not a session type}$$

$$\llceil T \times U \rrceil = \llceil T \rrceil \otimes \llceil U \rrceil \qquad \llceil \mathbf{1} \rrceil = \mathbf{1} \qquad \llceil T \multimap U \rrceil = \llceil T \rrceil^{\perp} \parr (\mathbf{1} \otimes \llceil U \rrceil)$$
$$\llceil T + U \rrceil = \llceil T \rrceil \oplus \llceil U \rrceil \qquad \llceil \mathbf{0} \rrceil = \mathbf{0} \qquad \llceil S \rrceil = \llfloor S \rrfloor^{\perp}$$

Translation on configurations and terms $\boxed{[\![\mathcal{C}]\!]^{\mathsf{c}}_r,\ [\![V]\!]^{\mathsf{v}}_r, \text{ and } [\![M]\!]^{\mathsf{m}}_r}$

$$[\![\circ\, M]\!]^{\mathsf{c}}_r = (\nu y y')([\![M]\!]^{\mathsf{m}}_y \parallel y'.y'[].\mathbf{0}) \qquad [\![(\nu x x')\mathcal{C}]\!]^{\mathsf{c}}_r = (\nu x x')[\![\mathcal{C}]\!]^{\mathsf{c}}_r \qquad [\![x \overset{z}{\leftrightarrow} y]\!]^{\mathsf{c}}_r = \bar{z}.z().x \leftrightarrow y$$
$$[\![\bullet\, M]\!]^{\mathsf{c}}_r = [\![M]\!]^{\mathsf{m}}_r \qquad [\![\mathcal{C} \parallel \mathcal{D}]\!]^{\mathsf{c}}_r = [\![\mathcal{C}]\!]^{\mathsf{c}}_r \parallel [\![\mathcal{D}]\!]^{\mathsf{c}}_r$$

$$[\![x]\!]^{\mathsf{v}}_r = r \leftrightarrow x \qquad [\![()]\!]^{\mathsf{v}}_r = r[].\mathbf{0} \qquad [\![\mathbf{inl}\ V]\!]^{\mathsf{v}}_r = r \triangleleft \mathrm{inl}.[\![V]\!]^{\mathsf{v}}_r$$
$$[\![\lambda x.M]\!]^{\mathsf{v}}_r = r(x).[\![M]\!]^{\mathsf{m}}_r \qquad [\![(V,W)]\!]^{\mathsf{v}}_r = r[x].([\![V]\!]^{\mathsf{v}}_x \parallel [\![W]\!]^{\mathsf{v}}_r) \qquad [\![\mathbf{inr}\ V]\!]^{\mathsf{v}}_r = r \triangleleft \mathrm{inr}.[\![V]\!]^{\mathsf{v}}_r$$

$$\begin{aligned}
&[\![V\ W]\!]^{\mathsf{m}}_r && = (\nu x x')(\nu y y')(y\langle x\rangle.r \leftrightarrow y \parallel [\![V]\!]^{\mathsf{v}}_{y'} \parallel [\![W]\!]^{\mathsf{v}}_{x'})\\
&[\![\mathbf{let}\ () = V\ \mathbf{in}\ M]\!]^{\mathsf{m}}_r && = (\nu x x')(x().[\![M]\!]^{\mathsf{m}}_r \parallel [\![V]\!]^{\mathsf{v}}_{x'})\\
&[\![\mathbf{let}\ (x,y) = V\ \mathbf{in}\ M]\!]^{\mathsf{m}}_r && = (\nu y y')(y(x).[\![M]\!]^{\mathsf{m}}_r \parallel [\![V]\!]^{\mathsf{v}}_{y'})\\
&[\![\mathbf{case}\ V\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\}]\!]^{\mathsf{m}}_r && = (\nu x x')(x \triangleright \{\mathrm{inl} : [\![M]\!]^{\mathsf{m}}_r; \mathrm{inr} : [\![N\{x/y\}]\!]^{\mathsf{m}}_r\} \parallel [\![V]\!]^{\mathsf{v}}_{x'})\\
&[\![\mathbf{absurd}\ V]\!]^{\mathsf{m}}_r && = (\nu x x')(x \triangleright \{\} \parallel [\![V]\!]^{\mathsf{v}}_{x'})\\
&[\![\mathbf{let}\ x = M\ \mathbf{in}\ N]\!]^{\mathsf{m}}_r && = (\nu x x')(x.[\![N]\!]^{\mathsf{m}}_r \parallel [\![M]\!]^{\mathsf{m}}_{x'})\\
&[\![V]\!]^{\mathsf{m}}_r && = \bar{r}.[\![V]\!]^{\mathsf{v}}_r
\end{aligned}$$

$$[\![\mathbf{link}]\!]^{\mathsf{v}}_r = r(y).y(x).\bar{r}.r().x \leftrightarrow y \qquad [\![\mathbf{send}]\!]^{\mathsf{v}}_r = r(y).y(x).y\langle x\rangle.\bar{r}.r \leftrightarrow y \qquad [\![\mathbf{wait}]\!]^{\mathsf{v}}_r = r(x).x().\bar{r}.r[].\mathbf{0}$$
$$[\![\mathbf{fork}]\!]^{\mathsf{v}}_r = r(x).\bar{r}.x\langle r\rangle.x.x[].\mathbf{0} \qquad [\![\mathbf{recv}]\!]^{\mathsf{v}}_r = r(x).x(y).\bar{r}.r\langle y\rangle.r \leftrightarrow x$$

Figure 8 Translation from HGV∗ to HCP.

encoding in HGV, *i.e.* $\mathbf{let}\ x = V\ \mathbf{in}\ M \longrightarrow_{\mathsf{M}} M\{V/x\}$.

$$\begin{array}{llcl}
\text{Terms} & L, M, N & ::= & V \mid \mathbf{let}\ x = M\ \mathbf{in}\ N \mid V\ W\\
& & \mid & \mathbf{let}\ () = V\ \mathbf{in}\ M \mid \mathbf{let}\ (x,y) = V\ \mathbf{in}\ M\\
& & \mid & \mathbf{absurd}\ V \mid \mathbf{case}\ V\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\}\\
\text{Values} & V, W & ::= & x \mid K \mid \lambda x.M \mid () \mid (V,W) \mid \mathbf{inl}\ V \mid \mathbf{inr}\ V\\
\text{Evaluation contexts} & E & ::= & \square \mid \mathbf{let}\ x = E\ \mathbf{in}\ M
\end{array}$$

We can *naively* translate HGV to HGV∗ ($(\!|\cdot|\!)$) by let-binding each subterm in a value position, *e.g.*, $(\!|\mathbf{inl}\ M|\!) = \mathbf{let}\ z = (\!|M|\!)\ \mathbf{in}\ \mathbf{inl}\ z$. Such a translation is given in Definition E.1; standard techniques can be applied if one wishes to avoid administrative redexes [40, 11].

HGV∗ to HCP The translation from HGV∗ to HCP is given in Figure 8. All control flow is encapsulated in values and let-bindings. We define a pair of translations on types, $\llfloor \cdot \rrfloor$ and $\llceil \cdot \rrceil$, such that $\llfloor T \rrfloor = \llceil T \rrceil^{\perp}$. We extend these translations pointwise to type environments and hyper-environments. We define translations on configurations ($[\![\cdot]\!]^{\mathsf{c}}_r$), terms ($[\![\cdot]\!]^{\mathsf{m}}_r$) and values ($[\![\cdot]\!]^{\mathsf{v}}_r$), where r is a fresh name denoting a special output channel over which the process sends a ping once it has reduced to a value, and then sends the value.

We translate an HGV sequent $\mathcal{G} \parallel \Gamma \vdash \mathcal{C} : T$ as $[\![\mathcal{C}]\!]^{\mathsf{c}}_r \vdash \llfloor \mathcal{G} \rrfloor \parallel \llfloor \Gamma \rrfloor, r : \mathbf{1} \otimes \llfloor T \rrfloor^{\perp}$, where Γ is the type environment corresponding to the main thread. The translation of a value $[\![V]\!]^{\mathsf{v}}_r$ immediately pings the output channel r to announce that it is a value. The translation of a let-binding $[\![\mathbf{let}\ w = M\ \mathbf{in}\ N]\!]^{\mathsf{m}}_r$ first evaluates M to a value, which then pings the internal channel x/x' and unblocks the continuation $x.[\![N]\!]^{\mathsf{m}}_r$.

▶ **Lemma 5.5** (Substitution). *If M is a well-typed term with $w \in \mathrm{fv}(M)$, and V is a well-typed value, then* $(\nu w w')([\![M]\!]^{\mathsf{m}}_r \parallel [\![V]\!]^{\mathsf{v}}_{w'}) \approx_\alpha [\![M\{V/w\}]\!]^{\mathsf{m}}_r$.

▶ **Theorem 5.6** (Operational Correspondence). *If $\mathcal{C}$ is a well-typed configuration:*

1. *if $\mathcal{C} \longrightarrow \mathcal{C}'$, then $[\![\mathcal{C}]\!]^{\mathsf{c}}_r \overset{\beta}{\Longrightarrow} [\![\mathcal{C}']\!]^{\mathsf{c}}_r$; and*
2. *if $[\![\mathcal{C}]\!]^{\mathsf{c}}_r \overset{\beta}{\longrightarrow} P$, then there exists a $\mathcal{C}'$ such that $\mathcal{C} \longrightarrow \mathcal{C}'$ and $P \approx [\![\mathcal{C}']\!]^{\mathsf{c}}_r$.*

Translating HCP to HGV We cannot translate HCP processes to HGV terms directly: HGV's term language only supports **fork** (see Appendix G for further discussion), so there is no way to translate an individual name restriction or parallel composition. We must first reunite each parallel composition with its corresponding name restriction, *i.e.*, translate to CP. We can then translate to HGV via known translations. Consequently, we factor the translation from HCP to HGV into three translations: (1) the translation from HCP into CP [28, Lemma 4.7]; (2) (a variant of) the translation from CP to GV [31, Figure 8]; and (3) the embedding of GV into HGV (Theorem 4.3). Translations (1) and (3) preserve and reflect reduction. However, Lindley and Morris's original translation from CP to GV preserves but does not reflect reduction due to an asynchronous encoding of choice. By adapting their translation to use a synchronous encoding of choice (Section 3), we obtain (2) a translation from CP to GV that both preserves and reflects reduction. Thus, composing all three translations together we obtain a translation from HCP to HGV that preserves and reflects reduction.

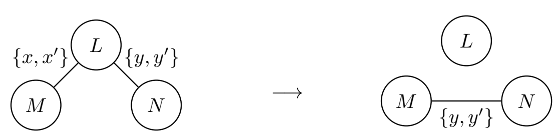
6 Extensions

In this section, we outline three extensions to HGV that exploit generalising the tree structure of processes to a forest structure. Full details are given in Appendix F. These extensions are of particular interest since HGV already supports a core aspect of forest structure, enabling its full utilisation merely through the addition of a structural rule. In contrast, to extend GV with forest structure one must distinguish two distinct introduction rules for parallel composition [31]. Other extensions to GV such as shared channels [31], polymorphism [33], and recursive session types [32] adapt to HGV almost unchanged.

From trees to forests The TC-Mix structural rule allows two type environments Γ_1, Γ_2 to be split by a hyper-environment separator *without* a channel connecting them. Mix [18] may be interpreted as concurrency *without* communication [31, 3].

$$\textsc{TC-Mix}\ \frac{\mathcal{G} \parallel \Gamma_1 \parallel \Gamma_2 \vdash \mathcal{C} : T}{\mathcal{G} \parallel \Gamma_1, \Gamma_2 \vdash \mathcal{C} : T}$$

A simpler link Consider threads $L = F[\textbf{link}\ (x, y)]$, M, N, where L connects to M by x and to N by y.



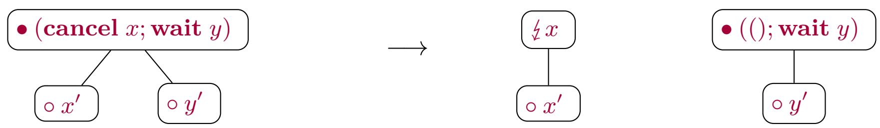
The result of link reduction has forest structure. Well-typed closed programs in both GV and HGV must *always* maintain tree structure. Different versions of GV do so in various unsatisfactory ways: one is pre-emptive blocking [31], which breaks confluence; another is two stage linking (Figure 4), which defers forwarding via a special link thread [32]. With TC-Mix, we can adjust the type schema for **link** to $(S \times \overline{S}) \multimap \mathbf{1}$ and use the following rule.

$$\textsc{E-Link-Mix} \qquad (\nu x x')(\mathcal{F}[\textbf{link}\ (x, y)] \parallel \phi N) \longrightarrow \mathcal{F}[()] \parallel \phi N\{y/x'\}$$

This formulation enables immediate substitution, maximimising concurrency.

Exceptions In order to support exceptions in the presence of linear endpoints [15, 35] we must have a way of *cancelling* an end point ($\textbf{cancel} : S \multimap \mathbf{1}$). Cancellation generates a special *zapper thread* ($\lightning x$) which severs a tree topology into a forest as in the following example.

$(\nu x x')(\nu y y')(\circ x' \parallel \circ y' \parallel \bullet(\textbf{cancel}\ x; \textbf{wait}\ y))$ $\qquad$ $(\nu x x')(\nu y y')(\circ x' \parallel \circ y' \parallel \lightning x \parallel \bullet((); \textbf{wait}\ y))$



7 Related work

Session Types and Functional Languages HGV traces its origins to a line of work initiated by Gay and collaborators [16, 48, 50, 17]. This family of calculi builds session types directly into a lambda calculus. Toninho et al. [47] take an alternative approach, stratifying their system into a session-typed process calculus and a separate functional calculus. There are many pragmatic embeddings of session type systems in existing functional programming languages [36, 41, 43, 22, 38, 25]. A detailed survey is given by Orchard & Yoshida [37].

Propositions as Sessions When Girard introduced linear logic [18] he suggested a connection with concurrency. Abramsky [1] and Bellin and Scott [5] give embeddings of linear logic proofs in π-calculus, where cut reduction is simulated by π-calculus reduction. Both embeddings interpret tensor as parallel composition. The correspondence with π-calculus is not tight in that these systems allow independent prefixes to be reordered. Caires and Pfenning [8] give a propositions as types correspondence between dual intuitionistic linear logic and a session-typed π-calculus called πDILL. They interpret tensor as output. The correspondence with π-calculus is tight in that independent prefixes may not be reordered. With CP [51], Wadler adapts πDILL to classical linear logic. Aschieri and Genco [2] give an interpretation of classical multiplicative linear logic as concurrent functional programs. They interpret $\parr$ as parallel composition, and the connection to session types is less direct.

Priority-based Calculi Systems such as πDILL, CP, and GV (and indeed HCP and HGV) ensure deadlock freedom by exploiting the type system to statically impose a tree structure on the communication topology — there can be at most one communication channel between any two processes. Another line of work explores a more liberal approach to deadlock freedom enabling some cyclic communication topologies, where deadlock freedom is guaranteed via *priorities*, which impose an order on actions. Priorites were introduced by Kobayashi and Padovani [24, 39] and adopted by Dardha and Gay [12] in Priority CP (PCP) and Kokke and Dardha in Priority GV (PGV) [26].

8 Conclusion and future work

HGV exploits hypersequents to resolve fundamental modularity issues with GV. As a consequence, we have obtained a tight operational correspondence between HGV and HCP. HGV is a modular and extensible core calculus for functional programming with *binary* session types. In future we intend to further exploit hypersequents in order to develop a modular and extensible core calculus for functional programming with *multiparty* session types. We would then hope to exhibit a similarly tight operational correspondence between this functional calculus and a multiparty variant of CP [10].

References

1 Samson Abramsky. Proofs as processes. 135(1):5–9, 1994.
2 Federico Aschieri and Francesco A. Genco. Par means parallel: multiplicative linear logic proofs as concurrent functional programs. *Proc. ACM Program. Lang.*, 4(POPL):18:1–18:28, 2020.
3 Robert Atkey, Sam Lindley, and J. Garrett Morris. Conflation confers concurrency. In *A List of Successes That Can Change the World*, volume 9600 of *Lecture Notes in Computer Science*, pages 32–55. Springer, 2016.
4 Arnon Avron. Hypersequents, logical consequence and intermediate logics for concurrency. 4:225–248, 1991.
5 Gianluigi Bellin and Philip J. Scott. On the pi-calculus and linear logic. 135(1):11–65, 1994.
6 Nick Benton and Andrew Kennedy. Exceptional syntax. 11(4):395–410, 2001.
7 Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theoretical Computer Science*, 195(2):205–226, 3 1998. `doi:10.1016/s0304-3975(97)00220-x`.
8 Luìs Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proc. of CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.
9 Marco Carbone, Ornela Dardha, and Fabrizio Montesi. Progress as compositional lock-freedom. In *Proc. of COORDINATION*, volume 8459 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 2014. `doi:10.1007/978-3-662-43376-8\_4`.
10 Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *CONCUR*, volume 59 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.
11 Olivier Danvy, Kevin Millikin, and Lasse R. Nielsen. On one-pass CPS transformations. *J. Funct. Program.*, 17(6):793–812, 2007.
12 Ornela Dardha and Simon J. Gay. A new linear logic for deadlock-free session-typed processes. In *Proc. of FoSSaCS*, volume 10803 of *LNCS*, pages 91–109. Springer, 2018.
13 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. 256:253–286, 2017.
14 Simon Fowler. *Typed Concurrent Functional Programming with Channels, Actors, and Sessions*. PhD thesis, 2019.
15 Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. 3(POPL):28:1–28:29, 2019.
16 Simon J. Gay and Rajagopal Nagarajan. Intensional and extensional semantics of dataflow programs. 15(4):299–318, 2003.
17 Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. 20(1):19–50, 2010.
18 Jean-Yves Girard. Linear logic. 50:1–102, 1987.
19 Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.
20 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proc. of ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.
21 Atsushi Igarashi, Peter Thiemann, Yuya Tsuda, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. 29:e17, 2019.
22 Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. Session type inference in Haskell. In *Proc. of PLACES*, volume 69 of *EPTCS*, pages 74–91, 2010. `doi:10.4204/EPTCS.69.6`.
23 Naoki Kobayashi. Type systems for concurrent programs. pages 439–453, 2003. `doi:10.1007/978-3-540-40007-3_26`.
24 Naoki Kobayashi. A new type system for deadlock-free processes. In *Proc. of CONCUR*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.

25 Wen Kokke and Ornela Dardha. Deadlock-free session types in linear Haskell. *CoRR*, abs/2103.14481, 2021. URL: `https://arxiv.org/abs/2103.14481`, `arXiv:2103.14481`.

26 Wen Kokke and Ornela Dardha. Prioritise the best variation. *CoRR*, abs/2103.14466, 2021. URL: `https://arxiv.org/abs/2103.14466`, `arXiv:2103.14466`.

27 Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Better late than never: A fully-abstract semantics for classical processes. 3(POPL), 2019.

28 Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Taking linear logic apart. In Thomas Ehrhard, Maribel Fernández, Valeria de Paiva, and Lorenzo Tortora de Falco, editors, *Proceedings Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Oxford, UK, 7-8 July 2018*, volume 292 of *Electronic Proceedings in Theoretical Computer Science*, pages 90–103. Open Publishing Association, 2019.

29 Jean-Jacques Lévy and Luc Maranget. Explicit substitutions and programming languages. In *Foundations of Software Technology and Theoretical Computer Science, 1999*, volume 1738 of *LNCS*. Springer, 1999. URL: `http://dx.doi.org/10.1007/3-540-46691-6_14`, `doi:10.1007/3-540-46691-6_14`.

30 Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. 185(2):182–210, 2003.

31 Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In Jan Vitek, editor, *Programming Languages and Systems*, pages 560–584. Springer Berlin Heidelberg, 2015.

32 Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. 51(9):434–447, 2016. `doi:10.1145/3022670.2951921`.

33 Sam Lindley and J. Garrett Morris. Lightweight functional session types. In Simon Gay and Antonio Ravara, editors, *Behavioural Types: from Theory to Tools*, chapter 12, pages 265–286. River publishers, 2017.

34 Fabrizio Montesi and Marco Peressotti. Classical transitions. Available on arXiv, 2018.

35 Dimitris Mostrous and Vasco T. Vasconcelos. Affine sessions. 14(4), 2018.

36 Matthias Neubauer and Peter Thiemann. An implementation of session types. In *Proc. of PADL*, volume 3057 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2004. `doi:10.1007/978-3-540-24836-1\_5`.

37 Dominic Orchard and Nobuko Yoshida. Session types with linearity in Haskell. *Behavioural Types: from Theory to Tools*, page 219, 2017.

38 Dominic A. Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *Proc. of POPL*, pages 568–581. ACM, 2016. `doi:10.1145/2837614.2837634`.

39 Luca Padovani. Deadlock and Lock Freedom in the Linear π-Calculus. In *Proc. of CSL-LICS*, pages 72:1–72:10. ACM, 2014.

40 Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

41 Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Proc. of Haskell*. ACM, 2008. `doi:10.1145/1411286.1411290`.

42 John C. Reynolds. The meaning of types—from intrinsic to extrinsic semantics. Technical Report RS-00-32, BRICS, 2000.

43 Matthew Sackman and Susan Eisenbach. Session types in Haskell: Updating message passing for the 21st century. 01 2008.

44 Davide Sangiorgi. π-calculus, internal mobility, and agent-passing calculi. *Theoretical Computer Science*, 167(1-2):235–274, 1996. `doi:10.1016/0304-3975(96)00075-8`.

45 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *Proc. of PARLE*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

46 Peter Thiemann and Vasco T. Vasconcelos. Label-dependent session types. 4(POPL):1–29, 2020.

47 Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 350–369. Springer, 2013.

48 Vasco Vasconcelos, Antonio Ravara, and Simon J. Gay. Session types for functional multithreading. In *CONCUR*, volume 3170 of *LNCS*, pages 497–511. Springer, 2004.
49 Vasco T. Vasconcelos. Fundamentals of session types. 217:52–70, 2012.
50 Vasco Thudichum Vasconcelos, Simon J. Gay, and Antonio Ravara. Type checking a multithreaded functional language with session types. 368(1-2):64–87, 2006.
51 Philip Wadler. Propositions as sessions. 24(2-3):384–418, 2014.

Appendices

A Omitted Definitions for Section 3: Hypersequent GV

A.1 Term Reduction

HGV values (U, V, W), evaluation contexts (E), and term reduction rules ($\longrightarrow_{\mathsf{M}}$) implement a standard call-by-value, left-to-right evaluation strategy. They are given in Figure 9.

Values and evaluation contexts

$$
\begin{array}{lrcl}
\text{Values} & U, V, W & ::= & K \mid \lambda x.M \mid () \mid (V, W) \mid \mathbf{inl}\ V \mid \mathbf{inr}\ V \\
\text{Evaluation contexts} & E & ::= & \square \\
 & & \mid & E\ M \mid V\ E \\
 & & \mid & \mathbf{let}\ () = E\ \mathbf{in}\ N \\
 & & \mid & (E, M) \mid (V, E) \mid \mathbf{let}\ (x, y) = E\ \mathbf{in}\ M \\
 & & \mid & \mathbf{inl}\ E \mid \mathbf{inr}\ E \mid \mathbf{case}\ E\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\}
\end{array}
$$

Term reduction $\boxed{M \longrightarrow_{\mathsf{M}} N}$

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

Figure 9 HGV, term reduction.

A.2 Choice

Internal and external choice are encoded with sum types and session delegation [23, 13]. Prior encodings of choice in GV [31] are pleasingly direct. External choice is implemented by receiving one of two possible session continuations, encoded as a sum type, and internal choice by forking a new thread to send such a value.

$$
\begin{array}{lll}
S \oplus S' & \triangleq & !(\overline{S_1} + \overline{S_2}).\mathbf{end}_! \\
S \,\&\, S' & \triangleq & ?(\overline{S_1} + \overline{S_2}).\mathbf{end}_? \\
\oplus\{\} & \triangleq & !\mathbf{0}.\mathbf{end}_! \\
\&\{\} & \triangleq & ?\mathbf{0}.\mathbf{end}_?
\end{array}
\qquad
\begin{array}{lll}
\mathbf{select}\ \ell & \triangleq & \lambda x.\mathbf{fork}\ (\lambda y.\mathbf{send}\ (\ell\ y, x)) \\
\mathbf{offer}\ L\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} & & \\
 & \triangleq & \begin{array}{l}\mathbf{let}\ (z, w) = \mathbf{recv}\ L\ \mathbf{in}\ \mathbf{wait}\ w; \\ \mathbf{case}\ z\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\}\end{array} \\
\mathbf{offer}\ L\ \{\} & \triangleq & \begin{array}{l}\mathbf{let}\ (z, w) = \mathbf{recv}\ L\ \mathbf{in}\ \mathbf{wait}\ w; \\ \mathbf{absurd}\ z\end{array}
\end{array}
$$

Alas, this encoding of internal choice is asynchronous. Consider the process below:

$$
(\nu x x')\left(\begin{array}{l} \circ\, \mathbf{let}\ x = \mathbf{select}\ \mathbf{inl}\ x\ \mathbf{in}\ \mathbf{let}\ y = \mathbf{send}\ ((), y)\ \mathbf{in}\ M \\ \parallel\ \bullet\, \mathbf{let}\ z = \mathbf{send}\ ((), z)\ \mathbf{in}\ \mathbf{offer}\ x'\ \{\mathbf{inl}\ x' \mapsto N_1; \mathbf{inr}\ x' \mapsto N_2\}\end{array}\right)
$$

The reader may be surprised that output on y may be visible before that on z. Surely, the select in the child thread must synchronise with the offer in the main thread? However, as **select** is implemented with **fork**, it returns immediately. As GV is confluent, such asynchrony cannot cause any observable difference in the results of a computation, but it is nevertheless unsatisfying from a concurrency perspective. To remedy the situation, we add a dummy synchronisation before exchanging the the sum type value, as follows:

$$
\begin{array}{lll}
S \oplus S' & \triangleq & !\mathbf{1}.!(\overline{S_1} + \overline{S_2}).\mathbf{end}_! \\
S \,\&\, S' & \triangleq & ?\mathbf{1}.?(S_1 + S_2).\mathbf{end}_? \\
\oplus\{\} & \triangleq & !\mathbf{1}.!\mathbf{0}.\mathbf{end}_! \\
\&\{\} & \triangleq & ?\mathbf{1}.?\mathbf{0}.\mathbf{end}_?
\end{array}
\qquad
\begin{array}{lll}
\mathbf{select}\ \ell & \triangleq & \lambda x.\left(\begin{array}{l}\mathbf{let}\ x = \mathbf{send}\ ((), x)\ \mathbf{in} \\ \mathbf{fork}\ (\lambda y.\mathbf{send}\ (\ell\ y, x))\end{array}\right) \\
\mathbf{offer}\ L\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} & & \\
 & \triangleq & \begin{array}{l}\mathbf{let}\ ((), z) = \mathbf{recv}\ L\ \mathbf{in}\ \mathbf{let}\ (w, z) = \mathbf{recv}\ z \\ \mathbf{in}\ \mathbf{wait}\ z; \mathbf{case}\ w\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\}\end{array} \\
\mathbf{offer}\ L\ \{\} & \triangleq & \begin{array}{l}\mathbf{let}\ ((), c) = \mathbf{recv}\ L\ \mathbf{in}\ \mathbf{let}\ (z, c) = \mathbf{recv}\ c \\ \mathbf{in}\ \mathbf{wait}\ c; \mathbf{absurd}\ z\end{array}
\end{array}
$$

B Abstract Process Structures

Due to space constraints, we have given the intuition behind abstract process structures in the main body of the paper. Here, we give the formal definitions and results.

Graph definitions. We begin by recalling the definition of an *undirected edge-labelled multigraph*: an undirected graph that allows multiple edges between vertices.

▶ **Definition B.1** (Undirected Multigraph). *An* undirected multigraph *G is a 3-tuple $(\mathcal{V}, \mathcal{E}, r)$ where:*

1. *$\mathcal{V}$ is a set of vertices*
2. *$\mathcal{E}$ is a set of edge names*
3. *r is a function $r : \mathcal{E} \mapsto \{\{v, w\} : v, w \in \mathcal{V}\}$ from edge names to an unordered pair of vertices*

Denote the size of a set as $|\cdot|$. A *path* is a sequence of edges connecting two vertices. A multigraph $G = (\mathcal{V}, \mathcal{E}, r)$ is *connected* if $|\mathcal{V}| = 1$, or if for every pair of vertices $v, w \in \mathcal{V}$ there is a path between v and w. A multigraph is *acyclic* if no path forms a cycle.

We define a *leaf* as a vertex connected to the remainder of a graph by a single edge.

▶ **Definition B.2** (Leaf). *Given an undirected multigraph $(\mathcal{V}, \mathcal{E}, r)$, a vertex $v \in \mathcal{V}$ is a* leaf *if there exists a single $e \in \mathcal{E}$ such that $v \in r(e)$.*

In an undirected tree containing at least two vertices, there must be at least two leaves.

▶ **Lemma B.3.** *If $G = (\mathcal{V}, \mathcal{E}, r)$ is an undirected tree where $|V| \geq 2$, then there exist at least two leaves in $\mathcal{V}$.*

Proof. For G to be an undirected tree where $|V| \geq 2$ and have fewer than two leaves, then there would need to be a cycle, contradicting acyclicity. ◀

Abstract process structures. An *abstract process structure* is a graph representation of a hyper-environment, where the vertices are typing environments and the edges are annotated by pairs of co-names.

Let $\mathsf{envs}(\Gamma_1 \parallel \cdots \parallel \Gamma_n) = \{\Gamma_1, \cdots, \Gamma_n\}$, and $|(\Gamma_1 \parallel \cdots \parallel \Gamma_n)| = n$.

Given a co-name set $\mathcal{N} = \{\{x_1, y_1\}, \ldots, \{x_n, y_n\}\}$, we can induce an abstract process structure on hyper-environments.

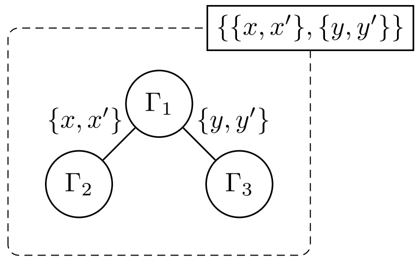
▶ **Definition B.4** (Abstract process structure). *The* abstract process structure of a hyper-environment $\mathcal{H}$ with respect to a co-name set $\mathcal{N} = \{\{x_1, y_1\}, \ldots, \{x_n, y_n\}\}$ *is an undirected multigraph $(\mathcal{V}, \mathcal{E}, r)$ defined as follows:*

1. $\mathcal{V} = \mathsf{envs}(\mathcal{H})$
2. $\mathcal{E} = \mathcal{N}$
3. $r = (\{x, y\} \mapsto \{\Gamma_1, \Gamma_2\})$ *for each* $\{x, y\} \in \mathcal{N}$ *such that* $\Gamma_1 \in \mathsf{envs}(\mathcal{H}), \Gamma_2 \in \mathsf{envs}(\mathcal{H}), x \in \mathrm{fv}(\Gamma_1), y \in \mathrm{fv}(\Gamma_2)$

▶ **Example B.5.** Suppose we have a hyper-environment $x : S_1, y : S_2 \parallel x' : \overline{S_1}, z : T \parallel y' : \overline{S_2}$ and suppose $\mathcal{N} = \{\{x, x'\}, \{y, y'\}\}$. Let $\Gamma_1 = x : S_1, y : S_2$; $\Gamma_2 = x' : \overline{S_1}, z : T$; and $\Gamma_3 = y' : \overline{S_2}$. The abstract process structure is defined as:

- $\mathcal{V} = \{\Gamma_1, \Gamma_2, \Gamma_3\}$
- $\mathcal{E} = \{\{x, x'\}, \{y, y'\}\}$
- $r(\{x, x'\}) \mapsto \{\Gamma_1, \Gamma_2\}$
  $r(\{y, y'\}) \mapsto \{\Gamma_1, \Gamma_3\}$

With the graphical representation:



▶ **Example B.6.** Let us consider another hyper-environment:

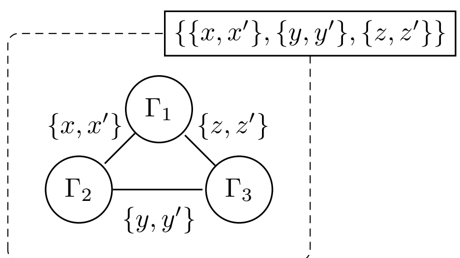
$$x : S_1, z' : \overline{S_3} \parallel x' : \overline{S_1}, y : S_2 \parallel y' : \overline{S_2}, z : S_3$$

and suppose $\mathcal{N} = \{\{x, x'\}, \{y, y'\}, \{z, z'\}\}$. Let $\Gamma_1 = x : S_1, z' : \overline{S_3}$; $\Gamma_2 = x' : \overline{S_1}, y : S_2$; and $\Gamma_3 = y' : \overline{S_2}, z : S_3$.

The APS is defined as:

- $\mathcal{V} = \{\Gamma_1, \Gamma_2, \Gamma_3\}$
- $\mathcal{E} = \{\{x, x'\}, \{y, y'\}, \{z, z'\}\}$
- $r(\{x, x'\}) \mapsto \{\Gamma_1, \Gamma_2\})$
  $r(\{y, y'\}) \mapsto \{\Gamma_2, \Gamma_3\}$
  $r(\{z, z'\}) \mapsto \{\Gamma_1, \Gamma_3\}$

With the graphical representation:

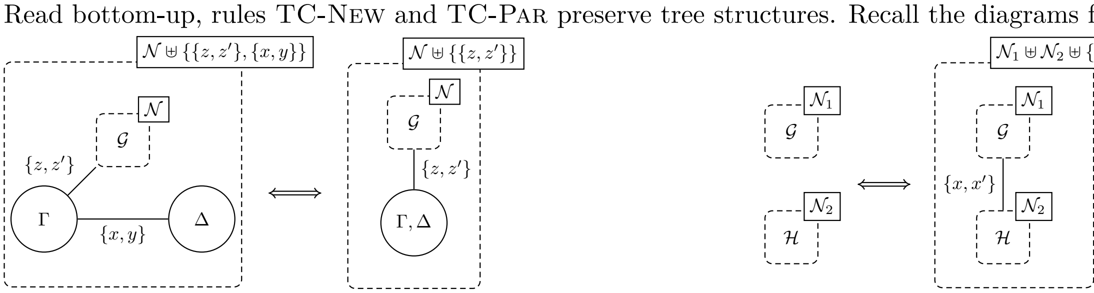


Note that Example B.5 forms a tree, whereas Example B.6 contains a cycle.

Only configurations typeable under a hyper-environment with a *tree structure* can be written in tree canonical form.

▶ **Definition B.7** (Tree structure)**.** *A hyper-environent $\mathcal{H}$ with names $\mathcal{N}$ has a* tree structure*, written* Tree$(\mathcal{H}, \mathcal{N})$*, if its APS is connected and acyclic.*

Read bottom-up, rules TC-New and TC-Par preserve tree structures. Recall the diagrams from Section 3:



The first diagram corresponds to TC-New. Reading left-to-right (and top-to-bottom in the case of the typing rule), since $\mathcal{G} \parallel \Gamma \parallel \Delta$ has tree structure, there must be some z, z' linking some sub-environment of $\mathcal{G}$ to either Γ or Δ (without loss of generality, we show the case for Γ). Given Γ is linked to Δ by an edge $\{x, y\}$, we can remove the edge and combine the vertices and retain a tree structure. Conversely, we can split some environment Γ, Δ into two nodes Γ and Δ, and connect them with a new edge representing a link between two variables.

The second diagram corresponds to TC-Par. Reading left-to-right (and top-to-bottom in the case of the typing rule), if we have two tree-structured abstract process structures for hyper-environments $\mathcal{G}$ (wrt. $\mathcal{N}_1$) and $\mathcal{H}$ (wrt. $\mathcal{N}_2$), one of which has a sub-environment containing x and the other has a sub-environment containing x' we can connect them by adding $\{x, x'\}$ to the name set. Conversely, given an APS linking $\mathcal{G}$ (defined wrt. $\mathcal{N}_1$) and $\mathcal{H}$ (defined wrt. $\mathcal{N}_2$) using an edge x, x', we know that $\mathcal{G}$ contains a sub-environment containing x, and $\mathcal{H}$ contains a sub-environment containing y. By removing the edge $\{x, x'\}$, we know that $\mathcal{G}$ has a tree structure wrt. $\mathcal{N}_1$, and $\mathcal{H}$ has a tree structure wrt. $\mathcal{N}_2$.

The following lemma states these intuitions formally. By analogy to Kleene equality, we write $\mathcal{P} \mathrel{\hat{\Longleftrightarrow}} \mathcal{Q}$, to mean that either $\mathcal{P}$ or $\mathcal{Q}$ is undefined, or $\mathcal{P} \iff \mathcal{Q}$.

▶ **Lemma B.8** (Tree structure)**.**

1. $\mathsf{Tree}((\mathcal{H} \parallel \Gamma_1, x_1 : S \parallel \Gamma_2, x_2 : \overline{S}), \mathcal{N} \uplus \{\{x_1, x_2\}\}) \mathrel{\hat{\Longleftrightarrow}} \mathsf{Tree}((\mathcal{H} \parallel \Gamma_1, \Gamma_2)), \mathcal{N})$
2. $\mathsf{Tree}((\mathcal{H}_1 \parallel \Gamma_1, x_1 : S), \mathcal{N}_1) \wedge \mathsf{Tree}((\mathcal{H}_2 \parallel \Gamma_2, x_2 : S), \mathcal{N}_2) \mathrel{\hat{\Longleftrightarrow}} \mathsf{Tree}((\mathcal{H}_1 \parallel \Gamma_1, x_1 : S \parallel \mathcal{H}_2 \parallel \Gamma_2, x_2 : \overline{S}), \mathcal{N}_1 \uplus \mathcal{N}_2 \uplus \{\{x_1, x_2\}\})$

Proof. We need only prove the cases where both sides of the bi-implication are defined.

Subcase (Clause 1).

Sub-subcase ($\Rightarrow$). Since $\mathcal{H} \parallel \Gamma_1, x_1 : S \parallel \Gamma_2, x_2 : \overline{S}$ has a tree structure wrt. $\mathcal{N} \uplus \{x_1, x_2\}$, we have that there exist y_1, y_2 such that $\{y_1, y_2\} \in \mathcal{N}$, where $y_1 \in \mathrm{fv}(\mathcal{H})$ and either $y_2 \in \mathrm{fv}(\Gamma_1)$ or $y_2 \in \mathrm{fv}(\Gamma_2)$. WLOG, assume $y_2 \in \mathrm{fv}(\Gamma_1)$. Since $\{x_1, x_2\} \in \mathcal{N}$ and the hyper-environment forms a tree structure, we know that $\{x_1, x_2\}$ is the only edge connecting Γ_1 and Γ_2, and we know that there is no edge connecting Γ_2 and $\mathcal{H}$. Thus, $\mathcal{H} \parallel \Gamma_1, \Gamma_2$ retains a tree structure.

Sub-subcase ($\Leftarrow$). By similar reasoning to the $\Rightarrow$ case, there exist y_1, y_2 such that $\{y_1, y_2\} \in \mathcal{N}$, and $y_1 \in \mathrm{fv}(\mathcal{H})$, and either $y_2 \in \mathrm{fv}(\Gamma_1)$ or $y_2 \in \mathrm{fv}(\Gamma_2)$. Since both sides of the bi-implication are defined, we have that $\{x_1, x_2\} \notin \mathcal{N}$, and $x_1 \notin \mathrm{fv}(\Gamma_1)$, and $x_2 \notin \mathrm{fv}(\Gamma_2)$, and Γ_1, Γ_2 are not connected by a self-edge. Without loss of generality, assume $y_2 \in \mathrm{fv}(\Gamma_1)$. Since $\{y_1, y_2\}$ connects $\mathcal{H}$ and Γ_1, and $\{x_1, x_2\}$ connects Γ_1 and Γ_2, the graph remains connected and acyclic, and therefore retains a tree structure.

Subcase (Clause 2).

Sub-subcase ($\Rightarrow$). Since the LHS is defined, we know that $\mathcal{N}_1$ and $\mathcal{N}_2$ are disjoint, and do not contain an edge $\{x_1, x_2\}$. The result therefore follows from the standard graph theoretic result that joining two trees (in our case by connecting Γ_1 and Γ_2) results in another tree.

Sub-subcase ($\Leftarrow$). Since $\{x_1, x_2\}$ is in the co-name set, we know that $\Gamma_1, x_1 : S$ and $\Gamma_2, x_2 : \overline{S}$ are connected only by x_1 and x_2. Since the RHS is defined, we know there are edges connecting $\mathcal{H}_1$ to Γ_1, and $\mathcal{H}_2$ to Γ_2. The result follows from the standard graph theoretic intuition that removing an edge from a tree results in two trees.

◀

C Omitted Proofs for Section 3: Hypersequent GV

▶ **Lemma C.1** (Subterm typeability). *Suppose* $\mathbf{D}$ *is a derivation of* $\Gamma_1, \Gamma_2 \vdash E[M] : T$*. There exists a type* U *and some subderivation* $\mathbf{D}'$ *of* $\mathbf{D}$ *concluding* $\Gamma_2 \vdash M : U$*, where the position of* $\mathbf{D}'$ *in* $\mathbf{D}$ *coincides with the position of the hole in* $\mathbf{D}$*.*

Proof. By induction on the structure of E. ◀

▶ **Lemma C.2** (Replacement, Evaluation Contexts). *If:*

- $\mathbf{D}$ *is a derivation of* $\Gamma_1, \Gamma_2 \vdash E[M] : T$
- $\mathbf{D}'$ *is a subderivation of* $\boldsymbol{D}$ *concluding* $\Gamma_2 \vdash M : U$
- *The position of* $\boldsymbol{D}'$ *in* $\boldsymbol{D}$ *corresponds to that of the hole in* E
- $\Gamma_3 \vdash N : U$
- Γ_1, Γ_3 *is defined*

then $\Gamma_1, \Gamma_3 \vdash E[N] : T$*.*

Proof. By induction on the structure of E. ◀

▶ **Lemma C.3** (Substitution). *If:*

1. $\Gamma_1, x : U \vdash M : T$
2. $\Gamma_2 \vdash N : U$
3. Γ_1, Γ_2 *is defined*

then $\Gamma_1, \Gamma_2 \vdash M\{N/x\} : T$*.*

Proof. By induction on the derivation of $\Gamma_1, x : U \vdash M : T$. ◀

▶ **Lemma C.4** (Preservation, $\longrightarrow_{\mathsf{M}}$). *If* $\Gamma \vdash M : T$ *and* $M \longrightarrow_{\mathsf{M}} N$*, then* $\Gamma \vdash N : T$*.*

Proof. A standard induction on the derivation of $\longrightarrow_{\mathsf{M}}$. ◀

Runtime type merging is commutative and associative. We make use of these properties implicitly in the remainder of the proofs.

▶ **Lemma C.5.**
1. $R_1 \sqcap R_2 \iff R_2 \sqcap R_1$
2. $R_1 \sqcap (R_2 \sqcap R_3) \iff (R_1 \sqcap R_2) \sqcap R_3$

Proof. Immediate from the definition of $\sqcap$. ◀

▶ **Lemma C.6** (Preservation ($\equiv$)). *If* $\mathcal{G} \vdash \mathcal{C} : R$ *and* $\mathcal{C} \equiv \mathcal{D}$*, then* $\mathcal{G} \vdash \mathcal{D} : R$*.*

Proof. We consider the cases for the equivalence axioms; the congruence cases are straightforward applications of the IH.

Case (SC-ParAssoc).

$\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}$

$$\dfrac{\mathcal{G}_1 \vdash \mathcal{C} : R_1 \qquad \dfrac{\mathcal{G}_2 \vdash \mathcal{D} : R_2 \qquad \mathcal{G}_3 \vdash \mathcal{E} : R_3}{\mathcal{G}_2 \parallel \mathcal{G}_3 \vdash \mathcal{D} \parallel \mathcal{E} : R_2 \sqcap R_3}}{\mathcal{G}_1 \parallel \mathcal{G}_2 \parallel \mathcal{G}_3 \vdash \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) : R_1 \sqcap R_2 \sqcap R_3} \iff \dfrac{\dfrac{\mathcal{G}_1 \vdash \mathcal{C} : R_1 \qquad \mathcal{G}_2 \vdash \mathcal{D} : R_2}{\mathcal{G}_1 \parallel \mathcal{G}_2 \vdash \mathcal{C} \parallel \mathcal{D} : R_1 \sqcap R_2} \qquad \mathcal{G}_3 \vdash \mathcal{E} : R_3}{\mathcal{G}_1 \parallel \mathcal{G}_2 \parallel \mathcal{G}_3 \vdash (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} : R_1 \sqcap R_2 \sqcap R_3}$$

Case (SC-ParComm).

$\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C}$

$$\frac{\mathcal{G} \vdash \mathcal{C} : R_1 \qquad \mathcal{H} \vdash \mathcal{D} : R_2}{\mathcal{G} \parallel \mathcal{H} \vdash \mathcal{C} \parallel \mathcal{D} : R_1 \sqcap R_2} \iff \frac{\mathcal{H} \vdash \mathcal{D} : U \qquad \mathcal{G} \vdash \mathcal{C} : T}{\mathcal{G} \parallel \mathcal{H} \vdash \mathcal{D} \parallel \mathcal{C} : R_1 \sqcap R_2}$$

Case (SC-NewComm).

$(\nu x x')(\nu y y')\mathcal{C} \equiv (\nu y y')(\nu x x')\mathcal{C}$

Two illustrative subcases:

Subcase (1).

$$\frac{\dfrac{\mathcal{G} \parallel \Gamma_1, x : S \parallel \Gamma_2, x' : \overline{S} \parallel \Gamma_3, y : S' \parallel \Gamma_4, y' : \overline{S'} \vdash \mathcal{C} : R}{\mathcal{G} \parallel \Gamma_1, x : S \parallel \Gamma_2, x' : \overline{S} \parallel \Gamma_3, \Gamma_4 \vdash (\nu y y')\mathcal{C} : R}}{\mathcal{G} \parallel \Gamma_1, \Gamma_2 \parallel \Gamma_3, \Gamma_4 \vdash (\nu x x')(\nu y y')\mathcal{C} : R} \iff \frac{\dfrac{\mathcal{G} \parallel \Gamma_1, y : S' \parallel \Gamma_2, y' : \overline{S'} \parallel \Gamma_3, x : S \parallel \Gamma_4, x' : \overline{S} \vdash \mathcal{C} : R}{\mathcal{G} \parallel \Gamma_1, y : S' \parallel \Gamma_2, y' : \overline{S'} \parallel \Gamma_3, \Gamma_4 \vdash (\nu x x')\mathcal{C} : R}}{\mathcal{G} \parallel \Gamma_1, \Gamma_2 \parallel \Gamma_3, \Gamma_4 \vdash (\nu y y')(\nu x x')\mathcal{C} : R}$$

Subcase (2).

$$\frac{\dfrac{\mathcal{G} \parallel \Gamma_1, x : S, y : S' \parallel \Gamma_2, y' : \overline{S'} \parallel \Gamma_3, x' : \overline{S} \vdash \mathcal{C} : R}{\mathcal{G} \parallel \Gamma_1, \Gamma_2, x : S \parallel \Gamma_3, x' : \overline{S} \vdash (\nu y y')\mathcal{C} : R}}{\mathcal{G} \parallel \Gamma_1, \Gamma_2, \Gamma_3 \vdash (\nu x x')(\nu y y')\mathcal{C} : R} \iff \frac{\dfrac{\mathcal{G} \parallel \Gamma_1, x : S, y : S' \parallel \Gamma_2, y' : \overline{S'} \parallel \Gamma_3, x' : \overline{S} \vdash \mathcal{C} : R}{\mathcal{G} \parallel \Gamma_1, \Gamma_3, y : S' \parallel \Gamma_2, y' : \overline{S'} \vdash (\nu x x')\mathcal{C} : R}}{\mathcal{G} \parallel \Gamma_1, \Gamma_2, \Gamma_3 \vdash (\nu y y')(\nu x x')\mathcal{C} : R}$$

Case (SC-NewSwap).

$(\nu x y)\mathcal{C} \equiv (\nu y x)\mathcal{C}$

Follows immediately since hyper-environments are treated as unordered.

Case (SC-ScopeExt).

$\mathcal{C} \parallel (\nu x y)\mathcal{D} \equiv (\nu x y)(\mathcal{C} \parallel \mathcal{D})$

(where $x, y \notin \mathrm{fv}(\mathcal{C})$)

$$\frac{\dfrac{\mathcal{G} \vdash \mathcal{C} : R_1 \qquad \mathcal{H} \parallel \Gamma_1, x : S \parallel \Gamma_2, y : \overline{S} \vdash \mathcal{D} : R_2}{\mathcal{G} \vdash \mathcal{C} : R_1 \qquad \mathcal{H} \parallel \Gamma_1, \Gamma_2 \vdash (\nu x y)\mathcal{D} : R_2}}{\mathcal{G} \parallel \mathcal{H} \parallel \Gamma_1, \Gamma_2 \vdash \mathcal{C} \parallel (\nu x y)\mathcal{D} : R_1 \sqcap R_2} \iff \frac{\dfrac{\mathcal{G} \vdash \mathcal{C} : R_1 \qquad \mathcal{H} \parallel \Gamma_1, x : S \parallel \Gamma_2, y : \overline{S} \vdash \mathcal{D} : R_2}{\mathcal{G} \parallel \mathcal{H} \parallel \Gamma_1, x : S \parallel \Gamma_2, y : \overline{S} \vdash \mathcal{C} \parallel \mathcal{D} : R_1 \sqcap R_2}}{\mathcal{G} \parallel \mathcal{H} \parallel \Gamma_1, \Gamma_2 \vdash (\nu x y)(\mathcal{C} \parallel \mathcal{D}) : R_1 \sqcap R_2}$$

Case (SC-LinkComm).

$x \overset{z}{\leftrightarrow} y \equiv y \overset{z}{\leftrightarrow} x$

Assumption:

$$\frac{}{x : S, y : \overline{S} \vdash x \overset{z}{\leftrightarrow} y : \circ}$$

By dualising both variables, we have that $x : \overline{S}, y : \overline{\overline{S}}$. Since duality is an involution, we can show $x : S, y : \overline{S} \iff x : \overline{S}, y : S$.
Thus:

$$\frac{}{y : S, x : \overline{S} \vdash y \overset{z}{\leftrightarrow} x : \circ}$$

The reasoning for the symmetric case is identical.

◀

▶ **Lemma C.7** (Preservation ($\longrightarrow$)). *If $\mathcal{G} \vdash \mathcal{C} : R$ and $\mathcal{C} \longrightarrow \mathcal{D}$, then $\mathcal{G} \vdash \mathcal{D} : R$.*

Proof. By induction on the derivation of $\mathcal{C} \longrightarrow \mathcal{D}$. Where there is a choice for ϕ, we prove the case for $\phi = \bullet$ and expand $\mathcal{T}[M]$ to $\bullet(E[M])$ for some evaluation context E; the other cases are similar.

Case (E-Reify-Fork).

$\bullet E[\mathbf{fork}\ V] \longrightarrow (\nu xy)(\bullet E[x] \parallel \circ V\ y)$

Assumption:

$$\dfrac{\Gamma \vdash E[\mathbf{fork}\ V] : T}{\Gamma \vdash \bullet E[\mathbf{fork}\ V] : T}$$

By Lemma C.1, there exist Γ_1, Γ_2, S such that $\Gamma = \Gamma_1, \Gamma_2$ and $\Gamma_1, \Gamma_2 \vdash E[\mathbf{fork}\ V] : T$ and:

$$\dfrac{\Gamma_2 \vdash V : S \multimap \mathbf{end}_!}{\Gamma_2 \vdash \mathbf{fork}\ V : \overline{S}}$$

By Lemma C.2:

$$\dfrac{\Gamma_1, x : \overline{S} \vdash E[x] : T}{\Gamma_1, x : \overline{S} \vdash \bullet E[x] : T}$$

By TM-App, $\Gamma_2, y : S \vdash V\ y : \mathbf{end}_!$ and so by TC-Child, $\Gamma_2, y : S \vdash V\ y : \circ$
Recomposing:

$$\dfrac{\dfrac{\dfrac{\Gamma_1, x : \overline{S} \vdash E[x] : T}{\Gamma_1, x : \overline{S} \vdash \bullet E[x] : T} \qquad \dfrac{\Gamma_2, y : S \vdash V\ y : \mathbf{end}_!}{\Gamma_2, y : S \vdash \circ(V\ y) : \circ}}{\Gamma_1, x : \overline{S} \parallel \Gamma_2, y : S \vdash \bullet E[x] \parallel \circ(V\ y) : T}}{\Gamma_1, \Gamma_2 \vdash (\nu xy)(\bullet E[x] \parallel \circ(V\ y) : T}$$

as required.

Case (E-Comm-Send).

$(\nu xy)(\bullet E[\mathbf{send}\ (V, x)] \parallel \circ E'[\mathbf{recv}\ y]) \longrightarrow (\nu xy)(\bullet E[x] \parallel \circ E'[(V, y)])$

Assumption:

$$\dfrac{\dfrac{\dfrac{\Gamma, x : S \vdash E[\mathbf{send}\ (V, x)] : U}{\Gamma, x : S \vdash \bullet E[\mathbf{send}\ (V, x)] : U} \qquad \dfrac{\Gamma', y : \overline{S} \vdash E'[\mathbf{recv}\ y] : \mathbf{end}_!}{\Gamma', y : \overline{S} \vdash \circ E'[\mathbf{recv}\ y] : \circ}}{\Gamma, x : S \parallel \Gamma', y : \overline{S} \vdash \bullet E[\mathbf{send}\ (V, x)] \parallel \circ E'[\mathbf{recv}\ y] : U}}{\Gamma, \Gamma' \vdash (\nu xy)(\bullet E[\mathbf{send}\ (V, x)] \parallel \circ E'[\mathbf{recv}\ y]) : U}$$

By Lemma C.1, there exist Γ_1, Γ_2, S such that $\Gamma = \Gamma_1, \Gamma_2$, and $\Gamma_1, \Gamma_2, x : S \vdash E[\mathbf{send}\ (V, x)] : U$ and:

$$\dfrac{\Gamma_2 \vdash V : T \qquad x : !T.S' \vdash x : !T.S'}{\Gamma_2, x : !T.S' \vdash \mathbf{send}\ (V, x) : S'}$$

With the knowledge that $S = {!T.S}$, we can refine our original derivation:

$$\dfrac{\dfrac{\dfrac{\Gamma_1, \Gamma_2, x : {!T.S'} \vdash E[\textbf{send}\ (V, x)] : U}{\Gamma_1, \Gamma_2, x : {!T.S'} \vdash \bullet E[\textbf{send}\ (V, x)] : U} \qquad \dfrac{\Gamma', y : {?T.\overline{S'}} \vdash E'[\textbf{recv}\ y] : \textbf{end}_!}{\Gamma', y : {?T.\overline{S'}} \vdash \circ E'[\textbf{recv}\ y] : \circ}}{\Gamma_1, \Gamma_2, x : {!T.S'} \parallel \Gamma', y : {?T.\overline{S'}} \vdash \bullet E[\textbf{send}\ (V, x)] \parallel \circ E'[\textbf{recv}\ y] : U}}{\Gamma_1, \Gamma_2, \Gamma' \vdash (\nu xy)(\bullet E[\textbf{send}\ (V, x)] \parallel \circ E'[\textbf{recv}\ y]) : U}$$

Again by Lemma C.1, we have that $\Gamma', y : {?T.\overline{S'}} \vdash E'[\textbf{recv}\ y] : \textbf{end}_!$ and:

$$\dfrac{y : {?T.\overline{S'}} \vdash y : {?T.\overline{S'}}}{y : {?T.\overline{S'}} \vdash \textbf{recv}\ y : T \times \overline{S'}}$$

We can show:

$$\dfrac{\Gamma_2 \vdash V : T \qquad y : \overline{S'} \vdash y : \overline{S'}}{\Gamma_2, y : \overline{S'} \vdash (V, y) : T \times \overline{S'}}$$

By Lemma C.2, we have that $\Gamma_2, \Gamma', y : \overline{S'} \vdash E'[(V, y)] : \overline{S'}$.
Recomposing:

$$\dfrac{\dfrac{\dfrac{\Gamma_1, x : S' \vdash E[x] : U}{\Gamma_1, x : S' \vdash \bullet E[x] : U} \qquad \dfrac{\Gamma_2, \Gamma', y : \overline{S'} \vdash E'[(V, y)] : \textbf{end}_!}{\Gamma_2, \Gamma', y : \overline{S'} \vdash \circ E'[(V, y)] : \circ}}{\Gamma_1, x : S' \parallel \Gamma_2, \Gamma', y : \overline{S'} \vdash \bullet E[x] \parallel \circ E'[(V, y)] : U}}{\Gamma_1, \Gamma_2, \Gamma' \vdash (\nu xy)(\bullet E[x] \parallel \circ E'[(V, y)]) : U}$$

as required.

Case (E-Comm-Close).

$$(\nu xy)(\mathcal{T}[\textbf{wait}\ x] \parallel \circ y) \longrightarrow \mathcal{T}[()]$$

Taking $\mathcal{T} = \bullet E$, assumption:

$$\dfrac{\dfrac{\dfrac{\Gamma, x : \textbf{end}_? \vdash E[\textbf{wait}\ x] : T}{\Gamma, x : \textbf{end}_? \vdash \bullet E[\textbf{wait}\ x] : T} \qquad \dfrac{\dfrac{}{y : \textbf{end}_! \vdash y : \textbf{end}_!}}{y : \textbf{end}_! \vdash \circ y : \circ}}{\Gamma, x : \textbf{end}_? \parallel y : \textbf{end}_! \vdash \bullet E[\textbf{wait}\ x] \parallel \circ y : T}}{\Gamma \vdash (\nu xy)(\bullet E[\textbf{wait}\ x] \parallel \circ y) : T}$$

By Lemma C.1, we have that:

$$\dfrac{x : \textbf{end}_? \vdash x : \textbf{end}_?}{x : \textbf{end}_? \vdash \textbf{wait}\ x : \mathbf{1}}$$

By Lemma C.2, $\Gamma \vdash E[()] : T$.
Recomposing:

$$\dfrac{\Gamma \vdash E[()] : T}{\Gamma \vdash \bullet E[()] : T}$$

as required.

Case (E-Reify-Link).

$$\mathcal{F}[\textbf{link}\ (x, y)] \longrightarrow (\nu z z')(x \overset{z}{\leftrightarrow} y \parallel \mathcal{F}[z'])$$

where z, z' fresh.
Taking $\mathcal{F} = \bullet E$, we have that:

$$\dfrac{\Gamma \vdash E[\textbf{link}\ (x, y)] : T}{\Gamma \vdash \bullet E[\textbf{link}\ (x, y)] : T}$$

By Lemma C.1, we have that $\Gamma = \Gamma', x : S, y : \overline{S}$ such that:

$$\dfrac{\dfrac{x : S \vdash x : S \qquad y : \overline{S} \vdash y : \overline{S}}{x : S, y : \overline{S} \vdash (x, y) : S \times \overline{S}}}{x : S, y : \overline{S} \vdash \textbf{link}\ (x, y) : \circ}$$

By Lemma C.2, we have that $\Gamma', z : \textbf{end}_{!} \vdash E[z] : T$.
Reconstructing:

$$\dfrac{\dfrac{\dfrac{}{z : \textbf{end}_{?}, x : S, y : \overline{S} \vdash x \overset{z}{\leftrightarrow} y : \circ} \qquad \Gamma', z : \textbf{end}_{!} \vdash \bullet E[z] : T}{z : \textbf{end}_{?}, x : S, y : \overline{S} \parallel \Gamma', z : \textbf{end}_{!} \vdash x \overset{z}{\leftrightarrow} y \parallel \bullet E[z] : T}}{\Gamma', x : S, y : \overline{S} \vdash (\nu z z')(x \overset{z}{\leftrightarrow} y \parallel \bullet E[z]) : T}$$

as required.

Case (E-Comm-Link).

$$(\nu z z')(\nu x x')(x \overset{z}{\leftrightarrow} y \parallel \circ z \parallel \bullet M) \longrightarrow \bullet(M\{y/x\})$$

Assumption:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : S, y : \overline{S}, z : \textbf{end}_{?} \vdash x \overset{z}{\leftrightarrow} y : \circ} \qquad \dfrac{\dfrac{z' : \textbf{end}_{!} \vdash z : \textbf{end}_{!}}{z' : \textbf{end}_{!} \vdash \circ z : \circ} \qquad \dfrac{\Gamma, x' : \overline{S} \vdash M : T}{\Gamma, x' : \overline{S} \vdash \bullet M : T}}{z' : \textbf{end}_{!} \parallel \Gamma, x' : \overline{S} \vdash \circ z \parallel \bullet M : T}}{x : S, y : \overline{S}, z : \textbf{end}_{?} \parallel z' : \textbf{end}_{!} \parallel \Gamma, x' : \overline{S} \vdash x \overset{z}{\leftrightarrow} y \parallel \circ z' \parallel \bullet M : T}}{\Gamma, y : \overline{S}, z : \textbf{end}_{?} \parallel z' : \textbf{end}_{!} \vdash (\nu x x')(x \overset{z}{\leftrightarrow} y \parallel \circ z' \parallel \bullet M) : T}}{\Gamma, y : \overline{S} \vdash (\nu z z')(\nu x x')(x \overset{z}{\leftrightarrow} y \parallel \circ z' \parallel \bullet M) : T}$$

By Lemma C.3, $\Gamma, y' : \overline{S} \vdash M\{y/x'\} : T$, thus:

$$\dfrac{\Gamma, y' : \overline{S} \vdash M\{y/x'\} : T}{\Gamma, y' : \overline{S} \vdash \bullet M\{y/x'\} : T}$$

as required.

Case (E-Res).

$$(\nu xy)\mathcal{C} \longrightarrow (\nu xy)\mathcal{D} \qquad \text{if } \mathcal{C} \longrightarrow \mathcal{D}$$

Immediate by the IH.

Case (E-Par).

$$\mathcal{C} \parallel \mathcal{D} \longrightarrow \mathcal{C}' \parallel \mathcal{D} \qquad \text{if } \mathcal{C} \longrightarrow \mathcal{C}'$$

Immediate by the IH.

Case (E-Equiv).

$$\mathcal{C} \longrightarrow \mathcal{D} \quad \text{if } \mathcal{C} \equiv \mathcal{C}', \mathcal{C}' \longrightarrow \mathcal{D}', \text{ and } \mathcal{D}' \equiv \mathcal{D}$$

Assumption: $\mathcal{G} \vdash \mathcal{C} : R$.
By Lemma C.6, $\mathcal{G} \vdash \mathcal{C}' : R$.
By the IH, $\mathcal{G} \vdash \mathcal{D}' : R$.
By Lemma C.6, $\mathcal{G} \vdash \mathcal{D} : R$, as required.

Case (E-Lift-M).

$$\phi M \longrightarrow \phi N \qquad \text{if } M \longrightarrow_{\mathsf{M}} N$$

Immediate by Lemma C.4.

◀

▶ **Theorem 3.2** (Preservation)**.**
1. *If* $\mathcal{G} \vdash \mathcal{C} : R$ *and* $\mathcal{C} \equiv \mathcal{D}$, *then* $\mathcal{G} \vdash \mathcal{D} : R$.
2. *If* $\mathcal{G} \vdash \mathcal{C} : R$ *and* $\mathcal{C} \longrightarrow \mathcal{D}$, *then* $\mathcal{G} \vdash \mathcal{D} : R$.

Proof. A direct corollary of Lemmas C.6 and C.7. ◀

C.1 Tree Canonical Forms

Recall that a configuration in tree canonical form if it is of the following form:

$$(\nu x_1 y_1)(\circ M_1 \parallel \cdots \parallel (\nu x_n y_n)(\circ M_n \parallel \phi N) \cdots)$$

where $x_i \in \mathrm{fn}(M_i)$ for each x_i, M_i.

Our technique for proving that any configuration typeable under a singleton hyper-environment can be written in tree canonical form is to demonstrate that the configuration typing rules induce a tree structure. Since undirected trees with at least two vertices must have at least two leaves, we can permute a child thread containing name x_i next to the binder $(\nu x_i y_i)$.

We can now prove that all configurations typeable under a single typing environment can be written in tree canonical form.

▶ **Theorem 3.5** (Tree canonical form)**.** *If* $\Gamma \vdash \mathcal{C} : R$, *then there exists some* $\mathcal{D}$ *such that* $\mathcal{C} \equiv \mathcal{D}$ *and* $\mathcal{D}$ *is in tree canonical form.*

Proof. By induction on the number of ν-binders in $\mathcal{C}$. In the case that $n = 0$, it must be the case that $\Gamma \vdash \phi M : R$ for some thread M, since parallel composition is only typeable under a hyper-environment containing two or more type environments. Therefore, $\mathcal{C}$ is in tree canonical form by definition.

In the case that $n \geq 1$, by Lemma C.6, we can rewrite the configuration as:

$$(\nu x_1 y_1) \cdots (\nu x_n y_n)(\circ M_1 \parallel \cdots \parallel \circ M_n \parallel \phi N)$$

Fix $\mathcal{N} = \{\{x_i, y_i\} \mid 1 \leq i \leq n\}$. By definition, Γ has a tree structure wrt. an empty co-name set. By repeated applications of TC-New, there exists some $\mathcal{G}$ such that $\mathcal{G} \vdash \circ M_1 \parallel \cdots \parallel \circ M_n \parallel \phi N : T$; by Lemma B.8 (clause 1, left-to-right), $\mathcal{G}$ has a tree structure.

Construct the APS for $\mathcal{G}$ using names $\mathcal{N}$; by Lemma B.3, there exist $\Gamma_1, \Gamma_2 \in \mathsf{envs}(\mathcal{H})$ such that Γ_1 and Γ_2 are leaves of the tree and therefore by the definition of the APS contain precisely one ν-bound name.

By TC-Par, there must exist two threads L_1, L_2 such that $\Gamma_1 \vdash L_1 : R_1$ and $\Gamma_2 \vdash L_2 : R_2$. By runtime type combination, at least one of R_1, R_2 must be $\circ$; without loss of generality assume this is R_1. Suppose (again without loss of generality) that the ν-bound name contained in Γ_1 is x_1 and $L_1 = M_1$.

Let $\mathcal{D} = (\nu x_2 y_2) \cdots (\nu x_n y_n)(\circ M_2 \parallel \cdots \parallel \circ M_n \parallel \phi N)$.

By Lemma C.6 and the fact that x_1 is the only ν-bound variable in M_1, we have that $\mathcal{C} \equiv (\nu x_1 y_1)(\circ M_1 \parallel \mathcal{D})$. By the IH, there exists some $\mathcal{D}'$ such that $\mathcal{D} \equiv \mathcal{D}'$ and $\mathcal{D}'$ is in canonical form. By construction we have that $\mathcal{C} \equiv (\nu x_1 y_1)(\circ M_1 \parallel \mathcal{D}')$, which is in tree canonical form as required. ◀

C.2 Progress

Let Ψ range over type environments where the type of each variable must be a session type:

$$\Psi ::= \cdot \mid \Psi, x : S$$

Functional reduction satisfies progress: under an environment only containing runtime names, a term will either reduce, be a value, or be ready to perform a communication action.

▶ **Lemma C.8** (Progress, Terms). *If $\Psi \vdash M : T$, then either there exists some N such that $M \longrightarrow_{\mathsf{M}} N$, or M can be written $E[N]$ for some $N \in \{\textbf{fork}\ V, \textbf{send}\ (V, W), \textbf{recv}\ V, \textbf{wait}\ V, \textbf{link}\ (V, W)\}$.*

Proof. A standard induction on the derivation of $\Psi \vdash M : T$. ◀

Note that tree canonical forms can be defined inductively:

$$\mathcal{CF} ::= \phi M \mid (\nu xy)(\mathcal{A} \parallel \mathcal{CF})$$

Lemma 3.8 follows as a direct corollary of a slightly more verbose property, which follows from the inductive definition of TCFs.

▶ **Definition C.9** (Open progress). *Suppose $\Psi \vdash \mathcal{C} : R$, where $\mathcal{C} \not\longrightarrow$, and $\mathcal{C}$ is in canonical form. We say that $\mathcal{C}$* satisfies open progress *if:*

1. $\mathcal{C} = (\nu x x')(\mathcal{A} \parallel \mathcal{D})$, *where:*
   - **a.** *There exist Ψ_1, Ψ_2 such that $\Psi = \Psi_1, \Psi_2$*
   - **b.** $\Psi_1, x : S \vdash \mathcal{A} : \circ$ *for some session type S, and* $\mathsf{blocked}(\mathcal{A}, y)$ *for some $y \in \mathrm{fv}(\Psi_1, x : S)$*
   - **c.** $\Psi_2, x' : \overline{S} \vdash \mathcal{D} : R$, *where $\mathcal{D}$ satisfies open progress*
2. $\mathcal{C} = \phi M$, *and either M is a value, there exist $z, z' \in \mathrm{fv}(\Psi)$, or* $\mathsf{blocked}(\phi M, x)$ *for some $x \in \mathrm{fv}(\Psi)$.*

▶ **Lemma C.10** (Open progress). *If $\Psi \vdash \mathcal{C} : R$ where $\mathcal{C}$ is in canonical form and $\mathcal{C} \not\longrightarrow$, then $\mathcal{C}$ satisfies open progress.*

Proof. By induction on the derivation of $\mathcal{G} \vdash \mathcal{C} : R$. By the definition of canonical forms, it must be the case that $\mathcal{C}$ is of the form $(\nu xy)(\mathcal{A} \parallel \mathcal{D})$ where $\mathcal{D}$ is in canonical form, or $\bullet M$.

We show the case where $\mathcal{C} = (\nu xy)(\circ M \parallel \mathcal{D})$; the case for $\mathcal{C} = \bullet M$ follows similar reasoning.

Assumption:

$$\dfrac{\dfrac{\Psi_1, x : S \vdash \mathcal{A} : \circ \qquad \Psi_2, y : \overline{S} \vdash \mathcal{D} : R}{\Psi_1, x : S \parallel \Psi_2, y : \overline{S} \vdash \mathcal{A} \parallel \mathcal{D} : R}}{\Psi_1, \Psi_2 \vdash (\nu xy)(\circ M \parallel \mathcal{D}) : R}$$

In both cases, by the induction hypothesis, $\Psi_2, y : \overline{S} \vdash \mathcal{D} : T$ satisfies open progress.

Subcase ($\mathcal{A} = \circ M$).
By Lemma C.8, either M is a value, or M can be written $E[N]$ for some communication and concurrency construct $N \in \{\textbf{fork } V, \textbf{send } (V, W), \textbf{recv } V, \textbf{wait } V, \textbf{link } (V, W)\}$.
Otherwise, M is a communication or concurrency construct. If $N = \textbf{fork } V$, then reduction could occur by E-Reify-Fork. If $N = \textbf{link } (V, W)$, then by the type schema for **link**, we have that $\textbf{link } (V, W)$ must be of the form $\textbf{link } (z, z')$ for $z, z' \in \text{fv}(\Psi, x : S)$ and could reduce by E-Reify-Link.
Otherwise, it must be the case that $\mathsf{blocked}(\circ M, y)$ for some $z \in \text{fv}(\Psi_1, x : S)$.
Thus, $(\nu xy)(\circ M \parallel \mathcal{D})$ satisfies open progress, as required.

Subcase ($\mathcal{A} = z_2 \overset{z_1}{\leftrightarrow} z_3$). We have that $z_1, z_2, z_3 \in \text{fv}(\Psi_1, x : S)$, and the thread must be blocked by definition.

◀

▶ **Lemma C.11** (Closed Progress). *Suppose $\Psi \vdash \mathcal{C} : R$ where $\mathcal{C} = (\nu x_1 y_1)(\mathcal{A}_1 \parallel \cdots \parallel (\nu x_n y_n)(\mathcal{A}_n \parallel \phi N) \cdots)$ is in tree canonical form. Either $\mathcal{C} \longrightarrow \mathcal{D}$ for some $\mathcal{D}$, or:*

1. *For each $\mathcal{A}_j$ for $1 \leq j \leq n$, $\mathsf{blocked}(\mathcal{A}_j, x_j)$*
2. *N is a value*

Proof. Since the environment is closed, by Lemma 3.8, for each $\mathcal{A}_j$ it must be that $\mathsf{blocked}(\mathcal{A}_j, z)$ for some $z \in \{y_i \mid i \in 1..j-1\} \cup \{x_j\}$.

Note that if two names x, y are co-names, and one thread is blocked on x, and another is blocked on y, then due to typing the names must be dual and reduction can occur.

Consider $\mathcal{A}_1$. Since the environment is closed, $\mathcal{A}_1$ must be blocked on x_1. Next, consider $\mathcal{A}_2$; the thread cannot be blocked on y_1 as reduction would occur. By the definition of TCFs, $\mathcal{A}_2$ must contain x_2 and by the typing rules cannot contain y_2, so the thread must be blocked on x_2. We can extend this argument to the remainder of the configuration. ◀

▶ **Theorem 3.10** (Global progress). *Suppose $\mathcal{C}$ is a ground configuration. Either there exists some $\mathcal{D}$ such that $\mathcal{C} \longrightarrow \mathcal{D}$, or $\mathcal{C} = \bullet V$ for some value V.*

Proof. By Lemma C.11, either $\mathcal{C}$ can reduce, or $\mathcal{C}$ can be written $(\nu x_1 y_1)(\circ \mathcal{A}_1 \parallel \cdots \parallel (\nu x_n y_n)(\circ \mathcal{A}_n \parallel \bullet V) \cdots)$ where $\mathsf{blocked}(\mathcal{A}_i, x_i)$ for each $\{x_i \mid i \in 1..n\}$.

Since $\mathcal{C}$ is ground, $\text{fv}(V) = \emptyset$. Consequently, due to acyclicity, no auxiliary thread can be blocked.

It follows that if $\mathcal{C} \not\longrightarrow$, then there cannot be any auxiliary threads and thus $\mathcal{C} = \bullet V$ for some value V. ◀

Determinism and Strong Normalisation HGV enjoys a strong form of determinism known as the diamond property, and due to linearity enjoys strong normalisation. Unlike with preservation and progress, the addition of hypersequents does not substantially change the arguments from [31].

▶ **Theorem C.12** (Diamond property). *If* $\mathcal{G} \vdash \mathcal{C} : T$*,* $\mathcal{C} \longrightarrow \mathcal{D}$*, and* $\mathcal{C} \longrightarrow \mathcal{D}'$*, then* $\mathcal{D} \equiv \mathcal{D}'$*.*

Proof. Similar to that of GV [31, 14]: $\longrightarrow_{\mathsf{M}}$ is deterministic, and due to linearity, any overlapping reductions are separate and may be performed in either order. ◀

▶ **Theorem C.13** (Termination). *If* $\mathcal{G} \vdash \mathcal{C} : T$*, there are no infinite sequences* $\mathcal{C} \longrightarrow \longrightarrow \cdots$*.*

Proof. As with GV [31, 14], due to linearity, HGV has an elementary strong normalisation proof. Let the size of a configuration be the sum of the sizes of all abstract syntax trees of all terms contained in threads. The size of a configuration is invariant under $\equiv$ and strictly decreases under $\longrightarrow$, so no infinite reduction sequences can exist. ◀

C.3 Derived typing rules for syntactic sugar

$$\textsc{T-Seq}\ \frac{\Gamma \vdash M : \mathbf{1} \qquad \Delta \vdash N : T}{\Gamma, \Delta \vdash M; N : T} \qquad \textsc{T-LamUnit}\ \frac{\Gamma \vdash M : T}{\Gamma \vdash \lambda().M : \mathbf{1} \multimap T} \qquad \textsc{T-LamPair}\ \frac{\Gamma, x : T, y : T' \vdash M : U}{\Gamma \vdash \lambda(x, y).M : T \times T' \multimap U}$$

$$\textsc{T-Let}\ \frac{\Gamma \vdash M : T \qquad \Delta, x : T \vdash N : U}{\Gamma, \Delta \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : U} \qquad \textsc{T-Select-Inl}\ \frac{}{\cdot \vdash \mathbf{select\ inl} : S \oplus S' \multimap S} \qquad \textsc{T-Select-Inr}\ \frac{}{\cdot \vdash \mathbf{select\ inr} : S \oplus S' \multimap S'}$$

$$\textsc{T-Offer}\ \frac{\Gamma \vdash L : S \,\&\, S' \qquad \Delta, x : S \vdash M : T \qquad \Delta, y : S' \vdash N : T}{\Gamma, \Delta \vdash \mathbf{offer}\ L\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} : T} \qquad \textsc{T-Offer-Absurd}\ \frac{\Gamma \vdash L : \&\{\}}{\Gamma, \Delta \vdash \mathbf{offer}\ L\ \{\} : T}$$

D Omitted Proofs for Section 4: Relation between HGV and GV

A simple embedding of GV into HGV. The simplest embedding of GV in HGV relies on the observation from Section 2 that each parallel composition splits a single channel, meaning that we can write an arbitrary closed GV configuration in the form:

$$\mathcal{C}_1 \parallel_{\langle x_1, y_1 \rangle} \cdots \parallel_{\langle x_{n-2}, y_{n-2} \rangle} \mathcal{C}_{n-1} \parallel_{\langle x_{n-1}, y_{n-1} \rangle} \mathcal{C}_n$$

where each $\mathcal{C}$ does not contain a further parallel composition, and any main thread is in $\mathcal{C}_n$. We can then subsequently embed the configuration in HGV as:

$$(\nu x_1 y_1)(\mathcal{C}_1 \parallel \cdots \parallel (\nu x_{n-2} y_{n-2})(\mathcal{C}_{n-2} \parallel (\nu x_{n-1} y_{n-1})(\mathcal{C}_{n-1} \parallel \mathcal{C}_n)) \cdots)$$

which is well-typed by construction. As a corollary, every well-typed, closed GV configuration is equivalent to a well-typed, closed HGV configuration.

A structure-preserving embedding of GV into HGV. Though the simple embedding of GV into HGV is sound, it does not respect the *intention* of GV. With a little care, we can provide a stronger result: every well-typed open GV configuration is exactly a well-typed HGV configuration. We proceed now with the proof of Theorem 4.3.

▶ **Theorem 4.3** (Typeability of GV configurations in HGV). *If $\Gamma \vdash_{\mathsf{GV}} \mathcal{C} : R$, then there exists some $\mathcal{G}$ such that $\mathcal{G}$ is a splitting of Γ and $\mathcal{G} \vdash \mathcal{C} : R$.*

Proof. By induction on the derivation of $\Gamma \vdash \mathcal{C} : R$.

Case (TG-New). Assumption:

$$\frac{\Gamma, \langle y, y' \rangle : S^{\sharp} \vdash_{\mathsf{GV}} \mathcal{C} : R}{\Gamma \vdash_{\mathsf{GV}} (\nu y y')\mathcal{C} : R}$$

Suppose $\Gamma = \langle x_1, x_1' \rangle : S_1^{\sharp}, \ldots, \langle x_n, x_n' \rangle : S_n^{\sharp}$ (for clarity, without loss of generality, we assume the absence of non-session variables. As these are simply split between environments, they can be added orthogonally).
By the IH, we have that there exists some hyper-environment $\mathcal{G}$ such that $\mathcal{G} \vdash \mathcal{C} : R$, where $\mathcal{G}$ is a splitting of $\Gamma, \langle y, y' \rangle : S^{\sharp}$.
Since $\mathcal{G}$ is a splitting of $\mathcal{C}$, we know that $y : S \in \mathcal{G}$ and $y' : \overline{S} \in \mathcal{G}$, and that $\mathcal{G}$ has a tree structure with respect to names $\{\{x_1, x_1'\}, \ldots, \{x_n, x_n'\}, \{y, y'\}\}$.
Since $\mathcal{G}$ has a tree structure, by definition we have that $\mathcal{G} = \mathcal{G}' \parallel \Gamma_1, y : S \parallel \Gamma_2, y' : \overline{S}$ for some $\mathcal{G}', \Gamma_1, \Gamma_2$, where $\mathcal{G}'$ has a tree structure.
By Lemma B.8 (clause 1, left-to-right), $\mathcal{G}' \parallel \Gamma_1, \Gamma_2$ has a tree structure with respect to names $\{\{x_1, x_1'\}, \ldots, \{x_n, x_n'\}\}$.
Thus, we can show:

$$\frac{\mathcal{G}' \parallel \Gamma_1, y : S \parallel \Gamma_2, y' : \overline{S} \vdash \mathcal{C} : R}{\mathcal{G}' \parallel \Gamma_1, \Gamma_2 \vdash (\nu y y')\mathcal{C} : R}$$

where $\mathcal{G}' \parallel \Gamma_1, \Gamma_2$ has a tree structure with respect to names $\{\{x_1, x_1'\}, \ldots, \{x_n, x_n'\}\}$ and is therefore a splitting of Γ, as required.

Case (TG-Connect$_1$). Assumption:

$$\frac{\Gamma_1, y : S \vdash_{\mathsf{GV}} \mathcal{C} : R_1 \qquad \Gamma_2, y' : \overline{S} \vdash_{\mathsf{GV}} \mathcal{D} : R_2}{\Gamma_1, \Gamma_2, \langle y, y' \rangle : S^\sharp \vdash_{\mathsf{GV}} \mathcal{C} \parallel \mathcal{D} : R_1 \sqcap R_2}$$

Suppose $\Gamma_1 = \langle x_1, x'_1 \rangle : S^\sharp_1, \ldots, \langle x_m, x'_m \rangle : S^\sharp_m$ and $\Gamma_2 = \langle x_{m+1}, x'_{m+1} \rangle : S^\sharp_{m+1}, \ldots, \langle x_n, x'_n \rangle : S^\sharp_n$. By the IH, there exist hyper-environments $\mathcal{G}, \mathcal{H}$ such that:

1. $\mathcal{G}$ is a splitting of $\Gamma_1, y : S$
2. $\mathcal{H}$ is a splitting of $\Gamma_2, y' : \overline{S}$
3. $\mathcal{G} \vdash_{\mathsf{GV}} \mathcal{C} : R_1$
4. $\mathcal{H} \vdash_{\mathsf{GV}} \mathcal{D} : R_2$

By the definition of splittings, $\mathcal{G}$ and $\mathcal{H}$ can be written $\mathcal{G} = \mathcal{G}' \parallel \Gamma'_1, y : S$ and $\mathcal{H} = \mathcal{H}' \parallel \Gamma'_2, y' : \overline{S}$ for some Γ'_1, Γ'_2. Furthermore, $\mathcal{G}$ has a tree structure with respect to $\{\{x_1, x'_1\}, \ldots, \{x_m, x'_m\}\}$ and $\mathcal{H}$ has a tree structure with respect to $\{\{x_{m+1}, x'_{m+1}\}, \ldots, \{x_n, x'_n\}\}$.
By Lemma B.8 (clause 2, left-to-right), $\mathcal{G}' \parallel \Gamma'_1, y : S \parallel \mathcal{H}' \parallel \Gamma'_2, y' : \overline{S}$ has a tree structure with respect to $\{\{x_1, x'_1\}, \ldots, \{x_n, x'_n\}, \{y, y'\}\}$ and therefore $\mathcal{G} \parallel \mathcal{H}$ is a splitting of $\Gamma_1, \Gamma_2, \langle y, y' \rangle : S^\sharp$.
Recomposing in HGV:

$$\frac{\mathcal{G} \vdash \mathcal{C} : R_1 \qquad \mathcal{H} \vdash \mathcal{D} : R_2}{\mathcal{G} \parallel \mathcal{H} \vdash \mathcal{C} \parallel \mathcal{D} : R_1 \sqcap R_2}$$

as required.

Case (TG-Connect$_2$). Similar to TG-Connect$_1$.

Case (TG-Child). Assumption:

$$\frac{\Gamma \vdash M : \mathbf{end}_!}{\Gamma \vdash_{\mathsf{GV}} \circ M : \circ}$$

Since we mandated that variables of type $S^\sharp$ cannot appear in terms, there are no names of type $S^\sharp$ in Γ. Therefore, the singleton hyper-environment Γ is a valid splitting, and so we can conclude by TC-Child in HGV.

Case (TG-Main). Similar to TG-Child.

◀

▶ **Lemma 4.6.** *Suppose $\Gamma \vdash \mathcal{C} : R$ where $\mathcal{C}$ is in tree canonical form. Then, $\Gamma \vdash_{GV} \mathcal{C} : R$.*

Proof. By induction on the number of ν-bound names.
In the case that $n = 0$, the result follows immediately by TG-Child or TG-Main.
In the case that $n \geq 1$, we have that $\Gamma = \Gamma_1, \Gamma_2$ for some Γ_1, Γ_2 and:

$$\frac{\dfrac{\Gamma_1, x : S \vdash \circ L : \circ \qquad \Gamma_2, y : \overline{S} \vdash \mathcal{D} : R}{\Gamma_1, x : S \parallel \Gamma_2, y : \overline{S} \vdash \circ L \parallel \mathcal{D} : R}}{\Gamma_1, \Gamma_2 \vdash (\nu x y)(\circ L \parallel \mathcal{D}) : R}$$

such that $\mathcal{D}$ is in tree canonical form. That $\Gamma_1, x : S \vdash \circ L : \circ$ follows by the definition of tree canonical forms, since $x \in \mathrm{fv}(L)$.

By the IH, $\Gamma_2, y : \overline{S} \vdash \mathcal{D} : R$ in GV.

Thus, we can write:

$$\dfrac{\dfrac{\Gamma_1, x : S \vdash \circ L : \circ \qquad \Gamma_2, y : \overline{S} \vdash \mathcal{D} : R}{\Gamma_1, \Gamma_2, \langle x, y\rangle : S^\sharp \vdash \circ L \parallel \mathcal{D} : R}}{\Gamma_1, \Gamma_2 \vdash (\nu xy)(\circ L \parallel \mathcal{D}) : R}$$

as required. ◀

E Omitted Proofs for Section 5: Relation between HGV and CP

E.1 Structural Congruence

Structural congruence for HCP processes $\boxed{P \equiv Q}$

$$x \leftrightarrow^{A} y \equiv y \leftrightarrow^{A^\perp} x \qquad P \parallel \mathbf{0} \equiv P \qquad P \parallel Q \equiv Q \parallel P \qquad P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$$

$$(\nu x x')(\nu y y')P \equiv (\nu y y')(\nu x x')P \qquad (\nu x y)P \equiv (\nu y x)P \qquad (\nu x y)(P \parallel Q) \equiv P \parallel (\nu x y)Q \quad \text{if } x, y \notin \mathrm{fv}(P)$$

E.2 Translating HGV to HCP

▶ **Definition E.1.** *We can naively translate HGV to HGV∗ as follows:*

$$
\begin{array}{lcl}
(\!|x|\!) & = & x \\
(\!|\lambda x.M|\!) & = & \lambda x.(\!|M|\!) \\
(\!|L\ M|\!) & = & \textbf{let}\ x = (\!|L|\!)\ \textbf{in let}\ y = (\!|M|\!)\ \textbf{in}\ x\ y \\
(\!|()|\!) & = & () \\
(\!|\textbf{let}\ () = L\ \textbf{in}\ M|\!) & = & \textbf{let}\ z = (\!|L|\!)\ \textbf{in let}\ () = z\ \textbf{in}\ (\!|M|\!) \\
(\!|(M, N)|\!) & = & \textbf{let}\ x = (\!|M|\!)\ \textbf{in let}\ y = (\!|N|\!)\ \textbf{in}\ (x, y) \\
(\!|\textbf{let}\ (x, y) = L\ \textbf{in}\ M|\!) & = & \textbf{let}\ z = (\!|L|\!)\ \textbf{in let}\ (x, y) = z\ \textbf{in}\ (\!|M|\!) \\
(\!|\textbf{inl}\ M|\!) & = & \textbf{let}\ z = (\!|M|\!)\ \textbf{in inl}\ z \\
(\!|\textbf{inr}\ M|\!) & = & \textbf{let}\ z = (\!|M|\!)\ \textbf{in inr}\ z \\
(\!|\textbf{case}\ L\ \{\textbf{inl}\ x \mapsto M;\ \textbf{inr}\ y \mapsto N\}|\!) & = & \textbf{let}\ z = (\!|L|\!)\ \textbf{in case}\ z\ \{\textbf{inl}\ x \mapsto (\!|M|\!);\ \textbf{inr}\ y \mapsto (\!|N|\!)\} \\
(\!|\textbf{absurd}\ L|\!) & = & \textbf{let}\ z = (\!|L|\!)\ \textbf{in absurd}\ z
\end{array}
$$

▶ **Lemma E.2.** *Translations of terms are guaranteed to only have τ-transitions and transitions on the dedicated output channel. Formally, if M is a term, then $[\![M]\!]_r \xrightarrow{\ell}$, where $\ell = \tau$ or $\ell = \ell_r$. Values only have transitions on the dedicated output channel. Formally, if V is a value, then $[\![M]\!]_r \xrightarrow{\ell_r}$.*

Proof. By induction on M. ◀

▶ **Definition E.3** (Process-contexts)**.** *A process-context $P[\,]$ is a process with a single hole, denoted $\square$. We extend the typing rules, LTS and typing rules to process-contexts. We write $P[\,] \vdash \mathcal{G}/\mathcal{H}$ to mean that $P[\,]$ is typed under hyper-environment $\mathcal{H}$ expecting a process typed under $\mathcal{G}$,* i.e. *if $Q \vdash \mathcal{G}$ then $P[Q] \vdash \mathcal{H}$.*

▶ **Definition E.4.** *A process P is* blocked on *x if it only has transitions $P \xrightarrow{\ell_x}$.*

We write $\mathrm{cn}(P)$ to refer to the set of all channel names in P.

▶ **Lemma E.5.** *If $P[\,]$ is a process-context with $z, w, w' \notin \mathrm{cn}(P[\,])$, and Q is a process blocked on w', then $(\nu w w')(P[z \leftrightarrow w] \parallel Q) \approx_\alpha P[Q\{z/w'\}]$.*

Proof. By induction on the process-context $P[\,]$.

Case ($\square$).

$$
\begin{array}{cll}
\multicolumn{3}{l}{(\nu w w')(z \leftrightarrow w \parallel Q)} \\
\xrightarrow{\alpha} & Q\{z/w'\} & \\
\sim & Q\{z/w'\} & \text{(by reflexivity)}
\end{array}
$$

Case $((\nu xy)P[\,])$.

$$\begin{array}{lll} & (\nu ww')((\nu xy)(P[z{\leftrightarrow}w]) \parallel Q) & \\ \sim & (\nu xy)(\nu ww')(P[z{\leftrightarrow}w] \parallel Q) & \text{(by Lemma 5.4)} \\ \approx & (\nu xy)(P[Q\{z/w'\}]) & \text{(by Lemma 5.4 and IH)} \end{array}$$

Case $(P[\,] \parallel R)$.

$$\begin{array}{lll} & (\nu ww')(P[z{\leftrightarrow}w] \parallel R \parallel Q) & \\ \sim & (\nu ww')(P[z{\leftrightarrow}w] \parallel Q) \parallel R & \text{(by Lemma 5.4)} \\ \approx & P[Q\{z/w'\}] \parallel R & \text{(by Lemma 5.4 and IH)} \end{array}$$

Case $(R \parallel P[\,])$.

$$\begin{array}{lll} & (\nu ww')(R \parallel P[z{\leftrightarrow}w] \parallel Q) & \\ \sim & R \parallel (\nu ww')(P[z{\leftrightarrow}w] \parallel Q) & \text{(by Lemma 5.4)} \\ \approx & R \parallel P[Q\{z/w'\}] & \text{(by Lemma 5.4 and IH)} \end{array}$$

Case $(\pi.P[\,])$. Since Q is blocked on w', the process $(\nu ww')(\pi.P[z{\leftrightarrow}w] \parallel Q)$ has only one transition,

$$(\nu ww')(\pi.P[z{\leftrightarrow}w] \parallel Q) \xrightarrow{\pi} (\nu ww')(P[z{\leftrightarrow}w] \parallel Q).$$

The process $\pi.P[Q\{z/w'\}]$ has only one transition, also with label π,

$$\pi.P[Q\{z/w'\}] \xrightarrow{\pi} P[Q\{z/w'\}].$$

The resulting processes are bisimilar by the induction hypothesis.

Case $(x \triangleright \{\mathrm{inl} : P[\,]; \mathrm{inr} : P'[\,]\})$. Since Q is blocked on w', the process $(\nu ww')(x \triangleright \{\mathrm{inl} : P[z{\leftrightarrow}w]; \mathrm{inr} : P'[z{\leftrightarrow}w]\} \parallel Q)$ has only two transitions,

$$(\nu ww')(x \triangleright \{\mathrm{inl} : P[z{\leftrightarrow}w]; \mathrm{inr} : P'[z{\leftrightarrow}w]\} \parallel Q) \xrightarrow{x \triangleright \mathrm{inl}} (\nu ww')(P[z{\leftrightarrow}w] \parallel Q)$$

and

$$(\nu ww')(x \triangleright \{\mathrm{inl} : P[z{\leftrightarrow}w]; \mathrm{inr} : P'[z{\leftrightarrow}w]\} \parallel Q) \xrightarrow{x \triangleright \mathrm{inr}} (\nu ww')(P'[z{\leftrightarrow}w] \parallel Q).$$

The process $x \triangleright \{\mathrm{inl} : P[Q\{z/w'\}]; \mathrm{inr} : P'[Q\{z/w'\}]\}$ has only two transitions, also with labels $x \triangleright \mathrm{inl}$ and $x \triangleright \mathrm{inr}$,

$$x \triangleright \{\mathrm{inl} : P[Q\{z/w'\}]; \mathrm{inr} : P'[Q\{z/w'\}]\} \xrightarrow{x \triangleright \mathrm{inl}} P[Q\{z/w'\}]$$

and

$$x \triangleright \{\mathrm{inl} : P[Q\{z/w'\}]; \mathrm{inr} : P'[Q\{z/w'\}]\} \xrightarrow{x \triangleright \mathrm{inr}} P'[Q\{z/w'\}].$$

The resulting processes are bisimilar by the induction hypothesis.

◀

▶ **Lemma 5.5** (Substitution). *If M is a well-typed term with $w \in \mathrm{fv}(M)$, and V is a well-typed value, then $(\nu ww')([\![M]\!]^{\mathsf{m}}_r \parallel [\![V]\!]^{\mathsf{v}}_{w'}) \approx_\alpha [\![M\{V/w\}]\!]^{\mathsf{m}}_r$.*

Proof. Immediately from Lemma E.5. ◀

▶ **Lemma E.6** (Operational Correspondence, Terms). *If M is a well-typed term:*

1. *if* $M \longrightarrow_{\mathsf{M}} M'$, *then* $[\![M]\!]^{\mathsf{m}}_r \overset{\beta}{\Longrightarrow} [\![M']\!]^{\mathsf{m}}_r$; *and*
2. *if* $[\![M]\!]^{\mathsf{m}}_r \overset{\beta}{\longrightarrow} P$, *then there exists an* M' *such that* $M \longrightarrow_{\mathsf{M}} M'$ *and* $P \approx [\![M']\!]^{\mathsf{m}}_r$.

Proof.

1. By induction on the reduction $M \longrightarrow_{\mathsf{M}} M'$.

Case (E-Lam). The following diagram commutes:

$$\begin{array}{ccc}
(\lambda x.M)\ V & \xrightarrow{\longrightarrow_{\mathsf{M}}} & M\{V/x\} \\
\downarrow [\![\cdot]\!]^{\mathsf{m}}_r & & \\
(\nu x x')(\nu y y')(y\langle x\rangle.r{\leftrightarrow}y \parallel y'(x).[\![M]\!]^{\mathsf{m}}_{y'} \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & & \downarrow [\![\cdot]\!]^{\mathsf{m}}_r \\
\downarrow \overset{\beta}{\longrightarrow}\overset{\alpha}{\longrightarrow} & & \\
(\nu x x')(\nu y y')(r{\leftrightarrow}y \parallel [\![M]\!]^{\mathsf{m}}_{y'} \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & & \\
\downarrow \overset{\alpha}{\longrightarrow} & & \\
(\nu x x')([\![M]\!]^{\mathsf{m}}_r \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & \overset{\approx_\alpha \text{ (by Lemma 5.5)}}{\rule{3cm}{0.4pt}} & [\![M\{V/x\}]\!]^{\mathsf{m}}_r
\end{array}$$

Case (E-Unit). The following diagram commutes:

$$\begin{array}{ccc}
\textbf{let}\ () = ()\ \textbf{in}\ M & \xrightarrow{\longrightarrow_{\mathsf{M}}} & M \\
\downarrow [\![\cdot]\!]_r & & \\
(\nu x x')(x().[\![M]\!]^{\mathsf{m}}_r \parallel x'[].\mathbf{0}) & & \downarrow [\![\cdot]\!]_r \\
\downarrow \overset{\beta}{\longrightarrow} & & \\
[\![M]\!]_r \parallel \mathbf{0} & \overset{\equiv}{\rule{3cm}{0.4pt}} & [\![M]\!]^{\mathsf{m}}_r
\end{array}$$

Case (E-Pair). The following diagram commutes:

$$\begin{array}{ccc}
\textbf{let}\ (x, y) = (V, W)\ \textbf{in}\ M & \xrightarrow{\longrightarrow_{\mathsf{M}}} & M\{V/x\}\{W/y\} \\
\downarrow [\![\cdot]\!]_r & & \\
(\nu y y')(y(x).[\![M]\!]^{\mathsf{m}}_r \parallel y'[x'].([\![V]\!]^{\mathsf{v}}_{x'} \parallel [\![W]\!]^{\mathsf{v}}_{y'})) & & \downarrow [\![\cdot]\!]_r \\
\downarrow \overset{\beta}{\longrightarrow} & & \\
(\nu y y')(\nu x x')([\![M]\!]_r \parallel [\![V]\!]^{\mathsf{v}}_{x'} \parallel [\![W]\!]^{\mathsf{v}}_{y'}) & \overset{\approx_\alpha \text{ (by Lemma 5.5)}}{\rule{3cm}{0.4pt}} & [\![M\{V/x\}\{W/y\}]\!]^{\mathsf{m}}_r
\end{array}$$

Case (E-Inl). The following diagram commutes:

$$\begin{array}{ccc}
\textbf{case}\ \textbf{inl}\ V\ \{\textbf{inl}\ x \mapsto M;\ \textbf{inr}\ y \mapsto N\} & \xrightarrow{\longrightarrow_{\mathsf{M}}} & M\{V/x\} \\
\downarrow [\![\cdot]\!]_r & & \\
(\nu x x')(x \triangleright \{\mathrm{inl} : [\![M]\!]^{\mathsf{m}}_r; \mathrm{inr} : [\![N\{x/y\}]\!]^{\mathsf{m}}_r\} \parallel x' \triangleleft \mathrm{inl}.[\![V]\!]^{\mathsf{v}}_{x'}) & & \downarrow [\![\cdot]\!]_r \\
\downarrow \overset{\beta}{\longrightarrow} & & \\
(\nu x x')([\![M]\!]_r \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & \overset{\approx_\alpha \text{ (by Lemma 5.5)}}{\rule{3cm}{0.4pt}} & [\![M\{V/x\}]\!]^{\mathsf{m}}_r
\end{array}$$

Case (E-Inr). As E-Inl.

Case (E-Let). The following diagram commutes:

$$\begin{array}{ccc}
\textbf{let}\ x = V\ \textbf{in}\ M & \xrightarrow{\longrightarrow_{\mathsf{M}}} & M\{V/x\} \\
\downarrow [\![\cdot]\!]_r & & \\
(\nu x x')(x.[\![M]\!]^{\mathsf{m}}_r \parallel \bar{x}'.[\![V]\!]^{\mathsf{v}}_{x'}) & & \downarrow [\![\cdot]\!]_r \\
\downarrow \overset{\beta}{\longrightarrow}\overset{\beta}{\longrightarrow} & & \\
(\nu x x')([\![M]\!]_r \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & \overset{\approx_\alpha \text{ (by Lemma 5.5)}}{\rule{3cm}{0.4pt}} & [\![M\{V/x\}]\!]^{\mathsf{m}}_r
\end{array}$$

Case (E-Lift). The induction hypothesis gives us the first commuting diagram, which we use, together with HGV's E-Lift and HCP's E-Lift-Res and E-Lift-Par, to show that the second diagram commutes:

$$\begin{array}{ccc} M & \xrightarrow{\longrightarrow_{\mathsf{M}}} & M' \\ \downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{m}}_r} & & \downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{m}}_r} \\ [\![M]\!]^{\mathsf{m}}_r & \xrightarrow{\stackrel{\beta}{\Longrightarrow}} & [\![M']\!]^{\mathsf{m}}_r \end{array} \qquad \begin{array}{ccc} \mathbf{let}\ x = E[M]\ \mathbf{in}\ N & \xrightarrow{\longrightarrow_{\mathsf{M}}} & \mathbf{let}\ x = E[M']\ \mathbf{in}\ N \\ \downarrow{\scriptstyle [\![\cdot]\!]_r} & & \downarrow{\scriptstyle [\![\cdot]\!]_r} \\ (\nu x x')(x.[\![N]\!]^{\mathsf{m}}_r \parallel [\![M]\!]^{\mathsf{m}}_{x'}) & \xrightarrow{\stackrel{\beta}{\Longrightarrow}} & (\nu x x')(x.[\![N]\!]^{\mathsf{m}}_r \parallel [\![M']\!]^{\mathsf{m}}_{x'}) \end{array}$$

2. By induction on M.

Case ($U\ V$). There are two well-typed cases for U: either $U = z$ for some z; or $U = \lambda x.M$ for some x and M. If $U = z$, we have $(\nu x x')(\nu y y')(y\langle x\rangle.r{\leftrightarrow}y \parallel z{\leftrightarrow}y' \parallel [\![V]\!]^{\mathsf{v}}_{x'}) \stackrel{\beta}{\not\longrightarrow}$, which contradicts our premise. Therefore, $U = \lambda x.M$. The only possible β-transition is the one in the following diagram:

$$\begin{array}{ccc} (\lambda x.M)\ V & \xrightarrow{\longrightarrow_{\mathsf{M}}} & M\{V/x\} \\ \downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{m}}_r} & & \\ (\nu x x')(\nu y y')(y\langle x\rangle.r{\leftrightarrow}y \parallel y'(x).[\![M]\!]^{\mathsf{m}}_{y'} \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & & \\ \downarrow{\scriptstyle \stackrel{\beta}{\longrightarrow}\stackrel{\alpha}{\longrightarrow}} & & \downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{m}}_r} \\ (\nu x x')(\nu y y')(r{\leftrightarrow}y \parallel [\![M]\!]^{\mathsf{m}}_{y'} \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & & \\ \downarrow{\scriptstyle \stackrel{\alpha}{\longrightarrow}} & & \\ (\nu x x')([\![M]\!]^{\mathsf{m}}_r \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & \overline{\approx_\alpha \text{ (by Lemma 5.5)}} & [\![M\{V/x\}]\!]^{\mathsf{m}}_r \end{array}$$

Hence, $M' = M\{V/x\}$.

Case ($\mathbf{let}\ () = U\ \mathbf{in}\ M$). There are two well-typed cases for U: either $U = z$ for some z; or $U = ()$. If $U = z$, we have $(\nu x x')(x().[\![M]\!]^{\mathsf{m}}_r \parallel x'{\leftrightarrow}z) \stackrel{\beta}{\not\longrightarrow}$, which contradicts our premise. Therefore, $U = ()$. The only possible β-transition is the one in the following diagram:

$$\begin{array}{ccc} \mathbf{let}\ () = ()\ \mathbf{in}\ M & \xrightarrow{\longrightarrow_{\mathsf{M}}} & M \\ \downarrow{\scriptstyle [\![\cdot]\!]_r} & & \\ (\nu x x')(x().[\![M]\!]^{\mathsf{m}}_r \parallel x'[].\mathbf{0}) & & \downarrow{\scriptstyle [\![\cdot]\!]_r} \\ \downarrow{\scriptstyle \stackrel{\beta}{\longrightarrow}} & & \\ [\![M]\!]_r \parallel \mathbf{0} & \overline{\quad\equiv\quad} & [\![M]\!]^{\mathsf{m}}_r \end{array}$$

Hence, $M' = M$.

Case ($\mathbf{let}\ (x, y) = U\ \mathbf{in}\ M$). There are two well-typed cases for U: either $U = z$ for some z, or $U = (V, W)$. If $U = z$, we have $(\nu y y')(y(x).[\![M]\!]^{\mathsf{m}}_r \parallel y'{\leftrightarrow}z) \stackrel{\beta}{\not\longrightarrow}$, which contradicts our premise. Therefore, $U = (V, W)$. The only possible β-transition is the one in the following diagram:

$$\begin{array}{ccc} \mathbf{let}\ (x, y) = (V, W)\ \mathbf{in}\ M & \xrightarrow{\longrightarrow_{\mathsf{M}}} & M\{V/x\}\{W/y\} \\ \downarrow{\scriptstyle [\![\cdot]\!]_r} & & \\ (\nu y y')(y(x).[\![M]\!]^{\mathsf{m}}_r \parallel y'[x'].([\![V]\!]^{\mathsf{v}}_{x'} \parallel [\![W]\!]^{\mathsf{v}}_{y'})) & & \downarrow{\scriptstyle [\![\cdot]\!]_r} \\ \downarrow{\scriptstyle \stackrel{\beta}{\longrightarrow}} & & \\ (\nu y y')(\nu x x')([\![M]\!]_r \parallel [\![V]\!]^{\mathsf{v}}_{x'} \parallel [\![W]\!]^{\mathsf{v}}_{y'}) & \overline{\approx_\alpha \text{ (by Lemma 5.5)}} & [\![M\{V/x\}\{W/y\}]\!]^{\mathsf{m}}_r \end{array}$$

Case ($\mathbf{case}\ U\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ x \mapsto N\}$). There are two well-typed cases for U: either $U = z$ for some z; or $U = \mathbf{inl}\ V$. If $U = z$, we have $(\nu x x')(x \triangleright \{\mathrm{inl} : [\![M]\!]^{\mathsf{m}}_r; \mathrm{inr} : [\![N\{x/y\}]\!]^{\mathsf{m}}_r\} \parallel x'{\leftrightarrow}z) \stackrel{\beta}{\not\longrightarrow}$, which contradicts our premise. Therefore, $U = \textbf{inl}\ V$. The only possible β-transition is the one in the following diagram:

$$\begin{array}{ccc}
\textbf{case inl}\ V\ \{\textbf{inl}\ x \mapsto M;\ \textbf{inr}\ y \mapsto N\} & \xrightarrow{\longrightarrow_{\mathsf{M}}} & M\{V/x\} \\
\big\downarrow{\scriptstyle [\![\cdot]\!]_r} & & \\
(\nu x x')(x \triangleright \{\mathrm{inl} : [\![M]\!]^{\mathsf{m}}_r; \mathrm{inr} : [\![N\{x/y\}]\!]^{\mathsf{m}}_r\} \parallel x' \triangleleft \mathrm{inl}.[\![V]\!]^{\mathsf{v}}_{x'}) & & \big\downarrow{\scriptstyle [\![\cdot]\!]_r} \\
\big\downarrow{\scriptstyle \xrightarrow{\beta}} & & \\
(\nu x x')([\![M]\!]_r \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & \overset{\approx_\alpha \text{ (by Lemma 5.5)}}{\rule{3em}{0.4pt}} & [\![M\{V/x\}]\!]^{\mathsf{m}}_r
\end{array}$$

Case ($\textbf{absurd}\ U$). There is only one well-typed case for U: $U = z$ for some z. However, $(\nu x x')(x \triangleright \{\} \parallel x' \leftrightarrow z) \overset{\beta}{\not\longrightarrow}$, which contradicts our premise.

Case ($\textbf{let}\ x = M\ \textbf{in}\ N$). There are two possible cases: either $M = V$; or $[\![M]\!]^{\mathsf{m}}_{x'} \xrightarrow{\beta} P$ for some P. If M is a value, the only possible β-transition is the one in the following diagram:

$$\begin{array}{ccc}
\textbf{let}\ x = V\ \textbf{in}\ M & \xrightarrow{\longrightarrow_{\mathsf{M}}} & M\{V/x\} \\
\big\downarrow{\scriptstyle [\![\cdot]\!]_r} & & \\
(\nu x x')(x.[\![M]\!]^{\mathsf{m}}_r \parallel \bar{x}'.[\![V]\!]^{\mathsf{v}}_{x'}) & & \big\downarrow{\scriptstyle [\![\cdot]\!]_r} \\
\big\downarrow{\scriptstyle \xrightarrow{\beta}\xrightarrow{\beta}} & & \\
(\nu x x')([\![M]\!]_r \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & \overset{\approx_\alpha \text{ (by Lemma 5.5)}}{\rule{3em}{0.4pt}} & [\![M\{V/x\}]\!]^{\mathsf{m}}_r
\end{array}$$

Otherwise, if $[\![M]\!]^{\mathsf{m}}_{x'} \xrightarrow{\beta} P$ for some P, the induction hypothesis gives us an M' such that $M \longrightarrow_{\mathsf{M}} M'$ and $P \approx [\![M']\!]^{\mathsf{m}}_r$. We apply HGV's E-Lift and HCP's E-Lift-Res and E-Lift-Par.

Case (V). We have $\bar{r}.[\![V]\!]^{\mathsf{v}}_r \overset{\beta}{\not\longrightarrow}$, which contradicts our premise.

◀

▶ **Theorem 5.6** (Operational Correspondence). *If $\mathcal{C}$ is a well-typed configuration:*

1. *if $\mathcal{C} \longrightarrow \mathcal{C}'$, then $[\![\mathcal{C}]\!]^{\mathsf{c}}_r \overset{\beta}{\Longrightarrow} [\![\mathcal{C}']\!]^{\mathsf{c}}_r$; and*
2. *if $[\![\mathcal{C}]\!]^{\mathsf{c}}_r \xrightarrow{\beta} P$, then there exists a $\mathcal{C}'$ such that $\mathcal{C} \longrightarrow \mathcal{C}'$ and $P \approx [\![\mathcal{C}']\!]^{\mathsf{c}}_r$.*

Proof.

1. By induction on the reduction $\mathcal{C} \longrightarrow \mathcal{C}'$.

   Case (E-Reify-Fork). The following diagram commutes:

$$\begin{array}{ccc}
F[\textbf{fork}\ (\lambda w.M)] & \xrightarrow{\longrightarrow} & (\nu x x')(F[x] \parallel \circ\, M\{x'/w\}) \\
\big\downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{c}}_r} & & \\
[\![F]\!]^{\mathsf{c}}_r[(\nu y y')(\nu z z')\big(z\langle y\rangle.v \leftrightarrow z \parallel z'(u).u\langle z'\rangle.u.u[].\mathbf{0} \parallel y'(w).[\![M]\!]^{\mathsf{m}}_{y'}\big)] & & \big\downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{c}}_r} \\
\big\downarrow{\scriptstyle \xrightarrow{\tau}+} & & \\
[\![F]\!]^{\mathsf{c}}_r[(\nu y y')(v \leftrightarrow w \parallel y.y[].\mathbf{0} \parallel [\![M]\!]^{\mathsf{m}}_{y'})] & \overset{\approx_\alpha}{\rule{3em}{0.4pt}} & (\nu x x')([\![F]\!]^{\mathsf{c}}_r[v \leftrightarrow x] \parallel (\nu y y')([\![M\{x'/w\}]\!]^{\mathsf{m}}_y \parallel y'.y'[].\mathbf{0}))
\end{array}$$

   The channel v is internal to $[\![F]\!]^{\mathsf{c}}_r$. The diagram is simplified: it uses the canonical form $\lambda z.M$ as opposed to the opaque value form V and creates the substitution $M\{x'/z\}$ as opposed to the application $V\ x'$. The final two terms are bisimilar by Lemma E.5.

Case (E-Reify-Link). The following diagram commutes:

$$
\begin{array}{ccc}
\circ\, E[\textbf{link}\ (x, y)] & \xrightarrow{\quad\longrightarrow\quad} & (\nu z z')(x \overset{z}{\leftrightarrow} y \parallel \circ\, E[z']) \\
\downarrow \llbracket\cdot\rrbracket^{\mathsf{c}}_{r} & & \\
\multicolumn{3}{c}{(\nu a a')(\llbracket E\rrbracket^{\mathsf{m}}_{r}[(\nu z z')(\nu w w')(w\langle z\rangle.v{\leftrightarrow}w \parallel w'(t).t(s).\bar{w}'.w'().s{\leftrightarrow}t \parallel z'\langle x\rangle.y{\leftrightarrow}z' \parallel \bar{a}'.a'[].\mathbf{0}))]} \\
\downarrow \xrightarrow{\tau}{}^{+} & & \downarrow \llbracket\cdot\rrbracket^{\mathsf{c}}_{r} \\
(\nu a a')(\llbracket E\rrbracket^{\mathsf{m}}_{r}[\bar{v}.v().x{\leftrightarrow}y] \parallel \bar{a}'.a'[].\mathbf{0}) & \xrightarrow{\approx_\alpha} & (\nu z z')(\bar{z}.z().x{\leftrightarrow}y \parallel (\nu a a')(\llbracket E[v{\leftrightarrow}z']\rrbracket^{\mathsf{m}}_{a} \parallel \bar{a}'.a'[].\mathbf{0}))
\end{array}
$$

The channel v is internal to $\llbracket E\rrbracket^{\mathsf{m}}_{r}$.

Case (E-Comm-Link).

$$
\begin{array}{ccc}
(\nu z z')(\nu x x')(x \overset{z}{\leftrightarrow} y \parallel \circ\, z' \parallel \phi\ M) & \xrightarrow{\quad\longrightarrow\quad} & \phi\ (M\{y/x'\}) \\
\downarrow \llbracket\cdot\rrbracket^{\mathsf{c}}_{r} & & \\
(\nu z z')(\nu x x')(\bar{z}.z().x{\leftrightarrow}y \parallel (\nu w w')(z'{\leftrightarrow}w \parallel w'.w'[].\mathbf{0}) \parallel \llbracket \phi\ M\rrbracket^{\mathsf{c}}_{r}) & & \downarrow \llbracket\cdot\rrbracket^{\mathsf{c}}_{r} \\
\downarrow \xrightarrow{\tau}{}^{+} & & \\
\llbracket \phi\ M\rrbracket^{\mathsf{c}}_{r}\{y/x'\} & \xrightarrow{\approx_\alpha} & \llbracket \phi\ M\{y/x'\}\rrbracket^{\mathsf{c}}_{r}
\end{array}
$$

Case (E-Comm-Send).

$$
\begin{array}{ccc}
(\nu x x')(F[\textbf{send}\ (V, x)] \parallel F'[\textbf{recv}\ x']) & \xrightarrow{\quad\longrightarrow\quad} & (\nu x x')(F[x] \parallel F'[(V, x')]) \\
\downarrow \llbracket\cdot\rrbracket^{\mathsf{c}}_{r} & & \\
(\nu x x')\left(\begin{array}{l} \llbracket F\rrbracket^{\mathsf{c}}_{r}[(\nu y y')(\nu z z')(z\langle y\rangle.u{\leftrightarrow}z \parallel z'(t).t(s).t\langle s\rangle.\bar{z}'.z'{\leftrightarrow}t \parallel y'[w].(\llbracket V\rrbracket^{\mathsf{v}}_{w} \parallel x{\leftrightarrow}y'))] \parallel \\ \llbracket F'\rrbracket^{\mathsf{c}}_{r}[(\nu y y')(\nu z z')(z\langle y\rangle.v{\leftrightarrow}z \parallel z'(s).s(t).\bar{z}'.z'\langle t\rangle.z'{\leftrightarrow}s \parallel x'{\leftrightarrow}y')] \end{array}\right) & & \\
\downarrow \xrightarrow{\tau}{}^{+} & & \downarrow \llbracket\cdot\rrbracket^{\mathsf{c}}_{r} \\
(\nu x x')(\llbracket F\rrbracket^{\mathsf{c}}_{r}[(x\langle w\rangle.\bar{u}.x{\leftrightarrow}u \parallel \llbracket V\rrbracket^{\mathsf{v}}_{w})] \parallel \llbracket F'\rrbracket^{\mathsf{c}}_{r}[x'(t).\bar{v}.v\langle t\rangle.v{\leftrightarrow}x']) & & \\
\downarrow \xrightarrow{\tau}{}^{+} & & (\nu x x')(\llbracket F\rrbracket^{\mathsf{c}}_{r}[\bar{u}.x{\leftrightarrow}u] \parallel \llbracket F'\rrbracket^{\mathsf{c}}_{r}[\bar{v}.v[w].(\llbracket V\rrbracket^{\mathsf{v}}_{w} \parallel v{\leftrightarrow}x')]) \\
(\nu x x')(\llbracket F\rrbracket^{\mathsf{c}}_{r}[(\bar{u}.u{\leftrightarrow}x \parallel \llbracket V\rrbracket^{\mathsf{v}}_{w})] \parallel \llbracket F'\rrbracket^{\mathsf{c}}_{r}[\bar{v}.v\langle w\rangle.v{\leftrightarrow}x']) & \xrightarrow{\approx_\alpha} &
\end{array}
$$

The channels u and v are internal to $\llbracket F\rrbracket^{\mathsf{c}}_{r}$ and $\llbracket F'\rrbracket^{\mathsf{c}}_{r}$, respectively.

Case (E-Comm-Close).

$$
\begin{array}{ccc}
(\nu x x)(\circ\, x \parallel F[\textbf{wait}\ x']) & \xrightarrow{\quad\longrightarrow\quad} & F[()] \\
\downarrow \llbracket\cdot\rrbracket^{\mathsf{c}}_{r} & & \\
(\nu x x)\left(\begin{array}{l} (\nu y y')(\bar{y}.x{\leftrightarrow}y \parallel y'.y'[].\mathbf{0}) \parallel \\ \llbracket F\rrbracket^{\mathsf{c}}_{r}[(\nu z z')(\nu w w')(w\langle z\rangle.v{\leftrightarrow}w \parallel w'(s).s().\bar{w}'.w'[].\mathbf{0} \parallel x'{\leftrightarrow}z')] \end{array}\right) & & \downarrow \llbracket\cdot\rrbracket^{\mathsf{c}}_{r} \\
\downarrow \xrightarrow{\tau}{}^{+} & & \\
\llbracket F\rrbracket^{\mathsf{c}}_{r}[\bar{v}.v[].\mathbf{0}] & \xrightarrow{=} & \llbracket F\rrbracket^{\mathsf{c}}_{r}[\bar{v}.v[].\mathbf{0}]
\end{array}
$$

The channel v is internal to $\llbracket F\rrbracket^{\mathsf{c}}_{r}$.

Case (E-Res).

$$\begin{array}{ccc} (\nu xy)\mathcal{C} & \xrightarrow{\quad\longrightarrow\quad} & (\nu xy)\mathcal{C}' \\ \downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{c}}_r} & & \downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{c}}_r} \\ (\nu xy)[\![\mathcal{C}]\!]^{\mathsf{c}}_r & \xrightarrow{\overset{\beta}{\Longrightarrow}(\text{IH})} & (\nu xy)[\![\mathcal{C}']\!]^{\mathsf{c}}_r \end{array}$$

Case (E-Par).

$$\begin{array}{ccc} \mathcal{C} \parallel \mathcal{D} & \xrightarrow{\quad\longrightarrow\quad} & \mathcal{C}' \parallel \mathcal{D} \\ \downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{c}}_r} & & \\ [\![\mathcal{C}]\!]^{\mathsf{c}}_r \parallel [\![\mathcal{D}]\!]^{\mathsf{c}}_r & & \downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{c}}_r} \\ \downarrow{\scriptstyle \overset{\beta}{\Longrightarrow}(\text{IH})} & & \\ [\![\mathcal{C}']\!]^{\mathsf{c}}_r \parallel [\![\mathcal{D}]\!]^{\mathsf{c}}_r & \xrightarrow{\quad = \quad} & [\![\mathcal{C}' \parallel \mathcal{D}]\!]^{\mathsf{c}}_r \end{array}$$

Case (E-Equiv).

$$\begin{array}{ccccccc} \mathcal{C} & \xrightarrow{\quad\equiv\quad} & \mathcal{C}' & \xrightarrow{\quad\longrightarrow\quad} & \mathcal{D}' & \xrightarrow{\quad\equiv\quad} & \mathcal{E} \\ \downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{c}}_r} & & & & & & \\ [\![\mathcal{C}]\!]^{\mathsf{c}}_r & & & & & & \\ \downarrow{\scriptstyle \approx_\alpha (\text{Lemma 5.4})} & & & & & & \downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{c}}_r} \\ [\![\mathcal{C}']\!]^{\mathsf{c}}_r & & & & & & \\ \downarrow{\scriptstyle \overset{\beta}{\Longrightarrow}(\text{IH})} & & & & & & \\ [\![\mathcal{D}']\!]^{\mathsf{c}}_r & & & \xrightarrow{\qquad\approx_\alpha (\text{Lemma 5.4})\qquad} & & & [\![\mathcal{E}]\!]^{\mathsf{c}}_r \end{array}$$

Case (E-Lift-M). The cases for $\phi = \bullet$ and $\phi = \circ$ are similar; here we show the case for $\bullet$.

$$\begin{array}{ccc} \bullet M & \xrightarrow{\quad\longrightarrow\quad} & \bullet N \\ \downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{c}}_r} & & \downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{c}}_r} \\ [\![M]\!]^{\mathsf{m}}_r & \xrightarrow{\overset{\beta}{\Longrightarrow}(\text{Lemma E.6})} & [\![N]\!]^{\mathsf{m}}_r \end{array}$$

2. By induction on C; as with Lemma E.6, the only reductions that can occur for each case are those specified in (1).

◀

F Extensions

F.1 Unconnected processes

The TC-Par rule allows two processes to be composed in parallel if they are typeable under separate hyper-environments. In a closed program, hyper-environment separators are introduced by TC-Res, meaning that each process must be connected by a channel.

We can loosen this restriction by adding the following structural rule:

$$\textsc{TC-Mix}\quad \frac{\mathcal{G} \parallel \Gamma_1 \parallel \Gamma_2 \vdash \mathcal{C} : T}{\mathcal{G} \parallel \Gamma_1, \Gamma_2 \vdash \mathcal{C} : T}$$

TC-Mix allows two type environments Γ_1, Γ_2 to be split by a hyper-environment separator *without* a channel connecting them, and is inspired by Girard's [18] Mix rule; in the concurrent setting, Mix can be interpreted as concurrency *without* communication [31, 3]. TC-Mix admits a much simpler treatment of **link** and provides a crucial ingredient for handling exceptional behaviour.

Atkey *et al.* [3] show that conflating the 1 and $\perp$ types in CP (which correspond respectively to the $\mathbf{end}_!$ and $\mathbf{end}_?$ types in GV) is logically equivalent to adding the Mix rule and a 0-Mix rule (used to type an empty process). It follows that in the presence of TC-Mix, we use self-dual **end** type; in the GV setting, by using a self-dual **end** type, we decouple closing a channel from process termination. We therefore refine the TC-Child rule and the type schema for **fork** to ensure that each child thread returns the unit value, and replace the **wait** constant with a **close** constant which eliminates an endpoint of type **end**.

$$\mathbf{fork} : (S \multimap \mathbf{1}) \multimap \overline{S} \qquad \mathbf{close} : \mathbf{end} \multimap \mathbf{1} \qquad \textsc{TC-Child}\ \frac{\Gamma \vdash M : \mathbf{1}}{\Gamma \vdash \circ M : \mathbf{1}} \qquad \textsc{E-Close}\ (\nu xy)(E[\mathbf{close}\ x] \parallel E'[\mathbf{close}\ y]) \longrightarrow E[()] \parallel E'[()]$$

Given TC-Mix, we might expect a term-level construct **spawn** : $(\mathbf{1} \multimap \mathbf{1}) \multimap \mathbf{1}$ which spawns a parallel thread without a connecting channel. We can encode such a construct using **fork** and **close** (assuming fresh x and y):

$$\mathbf{spawn}\ M \triangleq \mathbf{let}\ x = \mathbf{fork}(\lambda y.\mathbf{close}\ y;\ M)\ \mathbf{in}\ \ \mathbf{close}\ x$$

Assuming the encoded **spawn** is running in a main thread, after two reduction steps, we are left with the configuration:

$$\dfrac{\dfrac{\dfrac{\cdot \vdash M : \mathbf{1}}{\cdot \vdash \circ M : \circ}\ \textsc{TC-Child} \qquad \dfrac{\cdot \vdash M : \mathbf{1}}{\cdot \vdash \bullet() : \mathbf{1}}\ \textsc{TC-Main}}{\cdot \parallel \cdot \vdash \circ M \parallel \bullet() : \mathbf{1}}\ \textsc{TC-Par}}{\cdot \vdash \circ M \parallel \bullet() : \mathbf{1}}\ \textsc{TC-Mix}$$

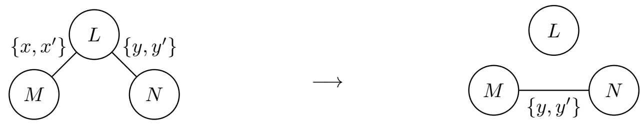
Note the essential use of TC-Mix to insert a hyper-environment separator.

The addition of TC-Mix does not affect preservation or progress. The result follows from routine adaptations of the proof of Theorem 3.2 and Theorem 3.10.

By relaxing the tree process structure restriction using TC-Mix, we can obtain a more efficient treatment of **link**, and can support the treatment of exceptions advocated by Fowler *et al.* [15].

F.2 A simpler link

The GV **link** (x, y) construct allows messages sent along x to be forwarded to y. Suppose we have three threads, L, M, N, where L holds endpoints x and y, connected to thread M over x and connected to N over y, and wishes to evaluate **link** (x, y):



Note that the process structure after the link takes place is a *forest* rather than a tree! Since well-typed, closed programs in both GV and HGV must *always* have a tree structure, different versions of GV have worked around this issue in slightly unsatisfactory ways.

Pre-emptive blocking. Lindley & Morris [31] implement **link** using the following rule (modified here to use a double-binder formulation):

$$(\nu x x')(\mathcal{F}[\mathbf{link}\ (x, y)] \parallel \mathcal{F}'[M]) \longrightarrow (\nu x x')(\mathcal{F}[x] \parallel \mathcal{F}'[\mathbf{wait}\ x'; M\{y/x'\}]) \qquad \text{where } x' \in \mathrm{fv}(M)$$

The first thread will eventually reduce to $\circ x$, at which point the second thread will synchronise to eliminate x and x' and then evaluate the continuation M with endpoint y substituted for x'. Unfortunately, this formulation of **link** pre-emptively inhibits reduction in the second thread, since the evaluation rule inserts a blocking **wait**. The resulting system does not satisfy the diamond property.

Link threads. HGV uses the incarnation of **link** advocated by [32], where linking is split into two stages: the first generates a fresh pair of endpoints z, z' and a link thread of the form $x \overset{z'}{\leftrightarrow} y$, and returns z to the calling thread. Once the calling thread has evaluated to a value (which must by typing be z), then the link substitution can take place. This formulation recovers confluence, but we still lose a degree of concurrency: communication on y is blocked until the linking thread has fully evaluated. In an ideal implementation, the behaviour of the linking thread would be irrelevant to the remainder of the configuration. The operation requires additional runtime syntax and thus complicates the metatheory.

With TC-Mix. The above issues are symptomatic of the fact that the process structure after a link takes place is a forest rather than a tree. However, with TC-Mix, we can refine the type schema for **link** to $(S \times \overline{S}) \multimap \mathbf{1}$ and we can use the following rule:

$$(\nu x x')(\mathcal{F}[\mathbf{link}\ (x, y)] \parallel \phi N) \longrightarrow \mathcal{F}[()] \parallel \phi N\{y/x'\}$$

This formulation has the strong advantage that the substitution takes place immediately and does not inhibit reduction. A variant of HGV replacing E-Reify-Link and E-Comm-Link with E-Link-Mix retains HGV's metatheory.

F.3 Exceptions

Mostrous & Vasconcelos [35] describe a process calculus allowing the *explicit cancellation* of a channel endpoint, accounting for exceptional scenarios such as a client disconnecting, or a thread encountering an unrecoverable error. Attempting to communicate with a cancelled endpoint raises an exception. Fowler *et al.* [15] extend these ideas to the functional setting, introducing Exceptional GV (EGV). EGV supports exceptional behaviour by adding:

- a new constant, **cancel** : $S \multimap \mathbf{1}$, which allows us to discard an arbitrary session endpoint with type S
- a construct **raise**, which raises an exception
- an exception handling construct **try** L **as** x **in** M **otherwise** N in the style of Benton & Kennedy [6], which attempts possibly-failing computation L, binding the result to x in success continuation M if successful and evaluating N if an exception is raised

As an example, consider the following two programs:

```
try
  let s = fork (λt.close(send (42, t)) in
  let (res, s) = recv s in
  close s; res as res in res
otherwise (−1)
```

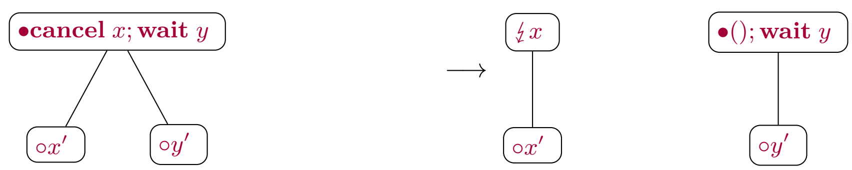
```
try
  let s = fork (λt.cancel t) in
  let (res, s) = recv s in
  close s; res as res in res
otherwise (−1)
```

In the first program, the child thread will send 42 to the parent thread, close its endpoint, and the exception handler will evaluate to 42. In the second program, instead of sending a value along t, the child thread discards its endpoint using **cancel**; the **recv** operation will then raise an exception and the exception handler will evaluate to -1.

Since hypersequents do not substantially change the operational semantics from the original presentation of EGV [15], we do not provide a full formal treatment here.

Why Mix? The runtime treatment of exceptional behaviour relies crucially on Mix. The key reason is that by explicitly discarding an endpoint, **cancel** generates a *zapper thread* which severs a tree process structure into a forest; future communications on the zapped name will trigger an exception. Consider the following example, where a thread cancels an endpoint x and then waits on an endpoint y.

$$(\nu x x')(\nu y y')(\circ x' \parallel \circ y' \parallel \bullet(\textbf{cancel}\ x; \textbf{wait}\ y)) \qquad (\nu x x')(\nu y y')(\circ x' \parallel \circ y' \parallel \lightning x \parallel \bullet((); \textbf{wait}\ y)$$



The configuration on the left has a tree process structure. However, after reduction, we obtain the configuration on the right which is clearly a forest and thus needs TC-Mix to be typable.

We have described a *synchronous* version of EGV. Extending our treatment to asynchrony as in the work of [15] is a routine adaptation.

G Hypersequents in term typing

Hypersequents allow us to cleanly separate name restriction and parallel composition in process configurations. Could we formulate a language HGV$^+$ which uses this technique at the *term level* to split **fork** into separate constructs for channel creation and thread creation? We argue splitting **fork** is more trouble than it's worth.

Suppose we extended term typing to allow hyper-environments, $\mathcal{G} \vdash M : T$, and introduced terms $\mathbf{let}\ \langle x, x' \rangle = \mathbf{new\ in}\ M$ and $\mathbf{let}\ \langle\rangle = \mathbf{spawn}\ M\ \mathbf{in}\ N$—which evaluate by simply creating a ν-binder and parallel composition, respectively—with the following typing rules:

$$\textsc{TM-LetNew}\quad \frac{\mathcal{G} \parallel \Gamma_1, x : S \parallel \Gamma_2, x' : \overline{S} \vdash M : T}{\mathcal{G} \parallel \Gamma_1, \Gamma_2 \vdash \mathbf{let}\ \langle x, x' \rangle = \mathbf{new\ in}\ M : T} \qquad \textsc{TM-LetSpawn}\quad \frac{\mathcal{G} \vdash M : \mathbf{end}_! \qquad \mathcal{H} \vdash N : T}{\mathcal{G} \parallel \mathcal{H} \vdash \mathbf{let}\ \langle\rangle = \mathbf{spawn}\ M\ \mathbf{in}\ N : T}$$

These rather ad-hoc rules mirror hypersequent cut and hypersequent composition: TM-LetNew creates a new channel with endpoints x and x', and requires them to be used in separate threads in the continuation M; and TM-LetSpawn takes a term M, spawns it as a child thread, and continues as N. Using these rules, we can encode $\mathbf{fork}\ M$ as $\mathbf{let}\ \langle x, x' \rangle = \mathbf{new\ in\ let}\ \langle\rangle = \mathbf{spawn}\ (M\ x)\ \mathbf{in}\ x'$.

Where else can we allow hyper-environments? In HCP, we have two options: (1) if we restrict *all logical rules* to singleton hypersequents and allow hyper-environments only in the rules for name restriction and parallel composition, we can use standard sequential semantics [34, 28]; but (2) if we allow hyper-environments in *any logical rules*, we must use a semantics which allows the corresponding actions to be delayed [27]. However, this is unlikely to be a property of logical rules, but rather due to the fact that the logical rules correspond exactly to the communication actions—which block reduction—and the structural rules to name restriction and parallel composition—which do not block reduction. Therefore, we expect the positions where hypersequents can safely occur to follow from the structure of evaluation contexts and whether any blocking term performs communication actions.

Regardless of our choice, we would be left with restrictions on the syntax of terms which seem sensible in a process calculus, but are surprising in a λ-calculus. In the strictest variant, where we disallow hyper-environments in all but the above two rules, uses of TM-LetNew and TM-LetSpawn may be interleaved, but no other construct may appear between a TM-LetNew and its corresponding TM-LetSpawn. Consider the following terms, where M uses x and y, and N uses x'. Term (1) may be well-typed, but (2) is always ill-typed:

$$\mathbf{let}\ y = 1\ \mathbf{in\ let}\ \langle x, x' \rangle = \mathbf{new\ in\ let}\ \langle\rangle = \mathbf{spawn}\ M\ \mathbf{in}\ N \tag{1}$$

$$\mathbf{let}\ \langle x, x' \rangle = \mathbf{new\ in\ let}\ y = 1\ \mathbf{in\ let}\ \langle\rangle = \mathbf{spawn}\ M\ \mathbf{in}\ N \tag{2}$$

Note that $\mathbf{let}\ \langle x, x' \rangle = \mathbf{new\ in}\ M$ is a single, monolithic term constructor—exactly what hypersequents were meant to prevent! However, if we attempt to decompose these constructors, we find that these are not the regular product and unit.