Logical Methods in Computer Science
Volume 19, Issue 3, 2023, pp. 3:1–3:52
https://lmcs.episciences.org/

Submitted Apr. 19, 2022
Published Jul. 12, 2023

SEPARATING SESSIONS SMOOTHLY

SIMON FOWLER [a], WEN KOKKE [b], ORNELA DARDHA [a], SAM LINDLEY [c], AND J. GARRETT MORRIS [d]

[a] University of Glasgow, UK
e-mail address: simon.fowler@glasgow.ac.uk, ornela.dardha@glasgow.ac.uk

[b] University of Strathclyde, UK
e-mail address: wen.kokke@strath.ac.uk

[c] The University of Edinburgh, UK
e-mail address: sam.lindley@ed.ac.uk

[d] The University of Iowa, USA
e-mail address: garrett-morris@uiowa.edu

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 an operational correspondence between HGV and HCP—a process calculus based on hypersequents and in a propositions-as-types correspondence with classical linear logic (CLL). 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 [Hon93, THK94, HVK98] are types used to model and 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 an unexpected message. 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 mainstream programming languages. This paper addresses two foundations for session types: (1) a session-typed concurrent lambda calculus called GV [LM15], intended to be a modular and extensible basis for functional programming languages with session types; and, (2) a session-typed process calculus called CP [Wad14], with a propositions-as-types correspondence to classical linear logic (CLL) [Gir87].

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 the 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

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.46298/LMCS-19(3:3)2023

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, since the expected transitions give rise to ill-typed terms. For example, we cannot write the expected transition rule for output, $x[y].(P \parallel Q) \xrightarrow{x[y]} P \parallel Q$, since $P \parallel Q$ is not a valid CP process. A similar issue arises when attempting to design a synchronisation transition rule for bound output (see [KMP19b] for a detailed discussion). Inspired by Carbone *et al.* [CMS18] who use hypersequents [Avr91] to give a logical grounding to choreographic programming languages [Mon13], Hypersequent CP (HCP) [KMP19a, KMP19b, MP18] restores the independence of these features by 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 [GV10], Lindley and Morris [LM15] 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 give a semantic characterisation of deadlocked processes, an extrinsic [Rey00] type system for configurations, and show that well-typed configurations are deadlock-free. There is, however, a large fly in this otherwise smooth ointment: GV's 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 [FLMD19].

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 resort to weak explicit substitutions [LM99] and only show that their translations between GV and CP preserve reduction. 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 [FLMD19].

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 an 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 why it is difficult to apply hyper-environments to term typing, Section 8 discusses related work, and Section 9 concludes and discusses future work.

This paper is an improved and extended version of a paper published at CONCUR 2021 [FKD+21]. Additional highlights include:

- a more detailed account of process structures;
- a more detailed account of extensions;
- a more detailed account of the metatheory for HCP; and
- a modified formulation of HCP's labelled transition system and the translation of **fork** in Section 5 fixing errors in the operational correspondence result from the CONCUR 2021 paper.

Proofs of all of the technical results are included in the paper.

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 chocolate bars and lollipops. If the vending machine is free, the customer can press ① to receive a chocolate bar or ② to receive a lollipop. If the vending machine is busy, the session ends.

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

The customer's session type is *dual*: where the vending machine sends a ChocolateBar, the customer receives a ChocolateBar, 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 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.

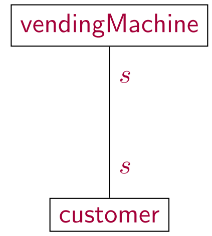
Lindley and Morris [LM15] introduce a semantics for GV, which evaluates programs embedded in process configurations, consisting of embedded programs, flagged as main ($\bullet$) or child ($\circ$) threads, ν-binders to create new channels, and parallel compositions:

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

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

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

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



(B) Process structure of Figure 1a.

Figure 1. Example program with acyclic process structure.

($\mathcal{C} \parallel (\nu x)\mathcal{C}' \equiv (\nu x)(\mathcal{C} \parallel \mathcal{C}')$ 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 $\mathcal{C} \parallel \mathcal{D}$, configurations $\mathcal{C}$ 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, they are *not* preserved by process equivalence. As Lindley and Morris write, (noting that their name restrictions bind *channels* rather than endpoint pairs, and their (νxy) abbreviates $(\nu x)(\nu y)$):

> 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 [LM15])**.** *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 [FLMD19] or admit deadlocks [ITT+19, TV20].

Note that CP only avoids the same issue through its combined $(\nu x)(P \parallel Q)$ term; attempts to split the term into a separate name restriction and parallel composition would also lose typability of equivalence.

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 $\boxed{\Gamma \vdash M : T}$

$$\text{TM-VAR}\ \frac{}{x : T \vdash x : T} \qquad \text{TM-CONST}\ \frac{}{\cdot \vdash K : T} \qquad \text{TM-LAM}\ \frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x.M : T \multimap U} \qquad \text{TM-APP}\ \frac{\Gamma \vdash M : T \multimap U \qquad \Delta \vdash N : T}{\Gamma, \Delta \vdash M\ N : U}$$

$$\text{TM-UNIT}\ \frac{}{\cdot \vdash () : \mathbf{1}} \qquad \text{TM-LETUNIT}\ \frac{\Gamma \vdash M : \mathbf{1} \qquad \Delta \vdash N : T}{\Gamma, \Delta \vdash \mathbf{let}\ () = M\ \mathbf{in}\ N : T} \qquad \text{TM-PAIR}\ \frac{\Gamma \vdash M : T \qquad \Delta \vdash N : U}{\Gamma, \Delta \vdash (M, N) : T \times U}$$

$$\text{TM-LETPAIR}\ \frac{\Gamma \vdash M : T \times T' \qquad \Delta, x : T, y : T' \vdash N : U}{\Gamma, \Delta \vdash \mathbf{let}\ (x, y) = M\ \mathbf{in}\ N : U} \qquad \text{TM-ABSURD}\ \frac{\Gamma \vdash M : \mathbf{0}}{\Gamma \vdash \mathbf{absurd}\ M : T} \qquad \text{TM-INL}\ \frac{\Gamma \vdash M : T}{\Gamma \vdash \mathbf{inl}\ M : T + U}$$

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

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 ($T + U$), 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 endpoints restrict process structure to *trees* [Wad14]; conflating them loosens this restriction to *forests* [ALM16]. 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; communication primitives K are given as constants. Each communication primitive K 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.

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

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

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

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

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

Figure 3. HGV, typing rules for configurations.

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*.

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 [Vas12] in which one name is bound to each endpoint of the channel. Link threads [LM16] 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 [LM15], exploiting *hypersequents* [Avr91] to recover modularity and extensibility. Inspired by HCP [MP18, KMP19b, KMP19a], configurations are typed under a *hyper-environment*, an unordered 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. Figure 4 gives the operational semantics for HGV, presented as a deterministic reduction relation on terms and a nondeterministic reduction relation on configurations. HGV values (U, V, W), evaluation contexts (E), and term reduction rules

Values and evaluation contexts

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

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

$$
\begin{array}{lll}
\text{E-LAM} & (\lambda x.M)\, V & \longrightarrow_{\mathsf{M}}\ M\{V/x\} \\
\text{E-UNIT} & \mathbf{let}\, () = ()\, \mathbf{in}\, M & \longrightarrow_{\mathsf{M}}\ M \\
\text{E-PAIR} & \mathbf{let}\, (x, y) = (V, W)\, \mathbf{in}\, M & \longrightarrow_{\mathsf{M}}\ M\{V/x, W/y\} \\
\text{E-INL} & \mathbf{case}\, \mathbf{inl}\, V\, \{\mathbf{inl}\, x \mapsto M;\ \mathbf{inr}\, y \mapsto N\} & \longrightarrow_{\mathsf{M}}\ M\{V/x\} \\
\text{E-INR} & \mathbf{case}\, \mathbf{inr}\, V\, \{\mathbf{inl}\, x \mapsto M;\ \mathbf{inr}\, y \mapsto N\} & \longrightarrow_{\mathsf{M}}\ N\{V/y\} \\
\text{E-LIFT} & E[M] & \longrightarrow_{\mathsf{M}}\ E[N],\ \text{if } M \longrightarrow_{\mathsf{M}} N
\end{array}
$$

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

$$
\begin{array}{llll}
\text{SC-PARASSOC} & \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} & \text{SC-PARCOMM} & \mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C} \\
\text{SC-NEWCOMM} & (\nu xy)(\nu zw)\mathcal{C} \equiv (\nu zw)(\nu xy)\mathcal{C} & \text{SC-NEWSWAP} & (\nu xy)\mathcal{C} \equiv (\nu yx)\mathcal{C} \\
\text{SC-SCOPEEXT} & (\nu xy)(\mathcal{C} \parallel \mathcal{D}) \equiv \mathcal{C} \parallel (\nu xy)\mathcal{D},\ \text{if } x, y \notin \mathrm{fv}(\mathcal{C}) & \text{SC-LINKCOMM} & x \overset{z}{\leftrightarrow} y \equiv y \overset{z}{\leftrightarrow} x
\end{array}
$$

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

$$
\begin{array}{lll}
\text{E-REIFY-FORK} & F[\mathbf{fork}\, V] & \longrightarrow\ (\nu x x')(F[x] \parallel \circ\, (V\, x')),\ \text{where } x, x' \text{ fresh} \\
\text{E-REIFY-LINK} & F[\mathbf{link}\, (x, y)] & \longrightarrow\ (\nu z z')(x \overset{z}{\leftrightarrow} y \parallel F[z']),\ \text{where } z, z' \text{ fresh}
\end{array}
$$

$$
\begin{array}{lll}
\text{E-COMM-LINK} & (\nu z z')(\nu x x')(x \overset{z}{\leftrightarrow} y \parallel \circ\, z' \parallel \phi\, M) & \longrightarrow\ \phi\, (M\{y/x'\}) \\
\text{E-COMM-SEND} & (\nu xy)(F[\mathbf{send}\, (V, x)] \parallel F'[\mathbf{recv}\, y]) & \longrightarrow\ (\nu xy)(F[x] \parallel F'[(V, y)]) \\
\text{E-COMM-CLOSE} & (\nu xy)(\circ\, y \parallel F[\mathbf{wait}\, x]) & \longrightarrow\ F[()]
\end{array}
$$

$$
\text{E-RES}\ \frac{\mathcal{C} \longrightarrow \mathcal{C}'}{(\nu xy)\mathcal{C} \longrightarrow (\nu xy)\mathcal{C}'}
\qquad
\text{E-PAR}\ \frac{\mathcal{C} \longrightarrow \mathcal{C}'}{\mathcal{C} \parallel \mathcal{D} \longrightarrow \mathcal{C}' \parallel \mathcal{D}}
\qquad
\text{E-EQUIV}\ \frac{\mathcal{C} \equiv \mathcal{C}' \quad \mathcal{C}' \longrightarrow \mathcal{D}' \quad \mathcal{D}' \equiv \mathcal{D}}{\mathcal{C} \longrightarrow \mathcal{D}}
\qquad
\text{E-LIFT-M}\ \frac{M \longrightarrow_{\mathsf{M}} M'}{F[M] \longrightarrow F[M']}
$$

FIGURE 4. HGV, operational semantics.

$(\longrightarrow_{\mathsf{M}})$ define a standard call-by-value, left-to-right evaluation strategy. A closed term either reduces to a value or is blocked on a communication action.

Thread contexts (F) extend evaluation contexts to threads. 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 configuration reduction relation gives a semantics for HGV's communication and concurrency constructs. 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 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.

Remark 3.2. Note that E-Comm-Link does not fire if z' is returned by a main thread. In closed configurations, typing ensures that such a configuration cannot arise: intuitively, a main thread can only obtain endpoints by **fork** or by receiving an endpoint.

Endpoints generated to communicate with forked threads (i.e., those passed to a child thread) will always have a session type terminating with $\mathbf{end}_?$, and a child thread cannot transmit an endpoint ending in $\mathbf{end}_!$, since the endpoint must be returned. Consequently, there is no way for a main thread to obtain endpoints with dual session types as required by the type of **link**. The case for open configurations is accounted for by our open progress result (see Section 3.1).

Choice. HGV does not include constructs for internal and external choice (for example, as shown in the vending machine example in Section 1). Internal and external choice are instead encoded with sum types and session delegation [Kob03, DGS17]. Prior encodings of choice in GV [LM15] are asynchronous. Instead, to encode synchronous choice we add a 'dummy' synchronisation before exchanging the value of sum type, as follows:

$$
\begin{aligned}
S \oplus S' &\triangleq \mathbf{!1}.!(\overline{S_1} + \overline{S_2}).\mathbf{end}_!\\
S \,\&\, S' &\triangleq \mathbf{?1}.?(S_1 + S_2).\mathbf{end}_?\\
\oplus\{\} &\triangleq \mathbf{!1}.\mathbf{!0}.\mathbf{end}_!\\
\&\{\} &\triangleq \mathbf{?1}.\mathbf{?0}.\mathbf{end}_?
\end{aligned}
$$

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

3.1. Metatheory.

HGV enjoys type preservation, deadlock freedom, confluence, and strong normalisation.

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

Theorem 3.3 (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. By induction on the derivations of $\mathcal{C} \equiv \mathcal{D}$ and $\mathcal{C} \longrightarrow \mathcal{D}$. See Appendix A. □

Before moving onto progress, we must introduce some technical machinery to allow us to reason about the structure of HGV programs.

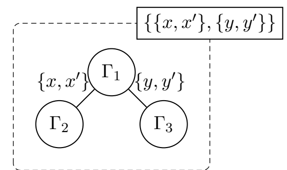
Abstract process structures. Unlike in GV, in HGV we cannot rely on the fact that exactly one channel is split over each parallel composition. Instead, we introduce the notion of an *abstract process structure* (APS). Abstract process structures are a crucial ingredient in showing that HGV configurations can be written in *tree canonical form*, which helps both with establishing progress results and also the correspondence between HGV and GV.

We begin by establishing the intuition behind the notion of an APS, and then describe the formal definitions. 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 $x_2 : \overline{S} \in \Gamma_2$.

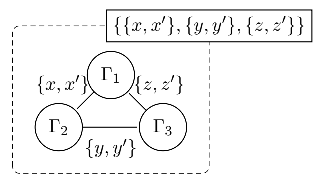
Example 3.4.

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.



Example 3.5.

Let $\mathcal{G} = \Gamma_1 \parallel \Gamma_2 \parallel \Gamma_3$, where $\Gamma_1 = x : S_1, z' : \overline{S_3}$, and $\Gamma_2 = x' : \overline{S_1}, y : S_2$, and $\Gamma_3 = y' : \overline{S_2}, z : S_3$, and suppose $\mathcal{N} = \{\{x, x'\}, \{y, y'\}, \{z, z'\}\}$. The APS for $\mathcal{G}$ and $\mathcal{N}$ is illustrated to the right.



Let us now discuss the formal definition of an APS. We begin by recalling the definition of an *undirected edge-labelled multigraph*: an undirected graph that allows multiple edges between vertices.

Definition 3.6 (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. A *leaf* is a vertex connected to the remainder of a graph by a single edge.

Definition 3.7 (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 3.8. *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. □

With the graph preliminaries in place, we are now ready to introduce the formal definition of an APS.

Definition 3.9 (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 $\Gamma_1 \in \mathsf{envs}(\mathcal{H}), \Gamma_2 \in \mathsf{envs}(\mathcal{H}), x \in \mathrm{fv}(\Gamma_1), y \in \mathrm{fv}(\Gamma_2)$

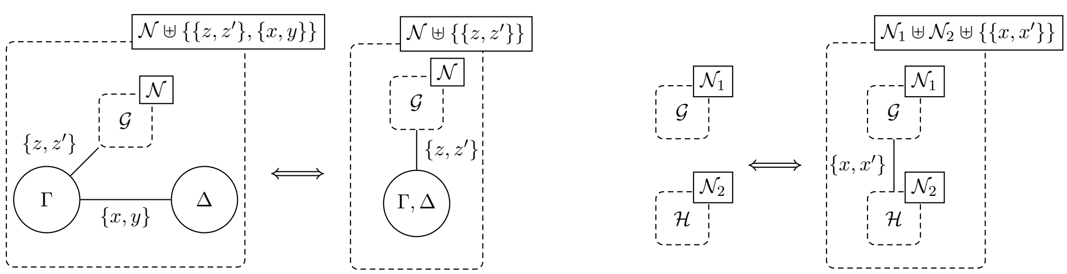
Example 3.10. The formal definition of the APS described in Example 3.4 is defined as:

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

Whereas Example 3.4 is a tree, Example 3.5 contains a cycle. Only configurations typeable under a hyper-environment with a *tree structure* can be written in tree canonical form.

Definition 3.11 (Tree structure)**.** A hyper-environment $\mathcal{H}$ with co-name set $\mathcal{N}$ has a *tree structure*, written $\mathsf{Tree}(\mathcal{H}, \mathcal{N})$, if its APS is connected and acyclic.

An HGV program $\bullet\, M$ has a single type environment, so is tree-structured; the same goes for child and link threads. 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. It follows that a configuration resulting from the reduction of an HGV program is also tree structured. Read bottom-up, TC-New and TC-Par preserve tree structure, which is illustrated by the following two pictures.



The following lemma states this intuition formally. By analogy to Kleene equality, we write $\mathcal{P} \stackrel{\simeq}{\Longleftrightarrow} \mathcal{Q}$, to mean that either $\mathcal{P}$ or $\mathcal{Q}$ is undefined, or $\mathcal{P} \iff \mathcal{Q}$.

Lemma 3.12 (Tree structure)**.**

- $\mathsf{Tree}((\mathcal{H} \parallel \Gamma_1, x_1 : S \parallel \Gamma_2, x_2 : \overline{S}), \mathcal{N} \uplus \{\{x_1, x_2\}\}) \stackrel{\simeq}{\Longleftrightarrow} \mathsf{Tree}((\mathcal{H} \parallel \Gamma_1, \Gamma_2)), \mathcal{N})$
- $\mathsf{Tree}((\mathcal{H}_1 \parallel \Gamma_1, x_1 : S), \mathcal{N}_1) \wedge \mathsf{Tree}((\mathcal{H}_2 \parallel \Gamma_2, x_2 : \overline{S}), \mathcal{N}_2) \stackrel{\simeq}{\Longleftrightarrow} \mathsf{Tree}((\mathcal{H}_1 \parallel \Gamma_1, x_1 : S \parallel \mathcal{H}_2 \parallel \Gamma_2, x_2 : \overline{S}), \mathcal{N}_1 \uplus \mathcal{N}_2 \uplus \{\{x_1, x_2\}\})$

Proof. By the definition of $\stackrel{\simeq}{\Longleftrightarrow}$, we need only consider the cases where both sides of the bi-implication are defined. Both results follow from the observation that adding an edge between two trees results in a tree, and removing an edge from a tree partitions the tree into two subtrees. ◻

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 3.17) and a means for embedding HGV in GV (Proposition 4.5).

Definition 3.13 (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 $1 \leq i \leq n$.

Every well-typed HGV configuration typeable under a single type environment can be written in tree canonical form.

Theorem 3.14 (Well-typed configurations in tree canonical forms)**.** *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 Theorem 3.3, 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 with respect to 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 3.12 (clause 1, right-to-left), $\mathcal{G}$ has a tree structure.

Construct the APS for $\mathcal{G}$ using names $\mathcal{N}$; by Lemma 3.8, 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 $\mathcal{C}_1, \mathcal{C}_2$ such that $\Gamma_1 \vdash \mathcal{C}_1 : R_1$ and $\Gamma_2 \vdash \mathcal{C}_2 : R_2$. 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 Theorem 3.3 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 induction hypothesis, 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. □

As hyper-environments capture parallelism, a configuration $\mathcal{C}$ typeable under hyper-environment $\Gamma_1 \parallel \cdots \parallel \Gamma_n$ is equivalent to n independent parallel processes.

Proposition 3.15 (Independence)**.** *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.*

Proof. By induction on the derivation of $\Gamma_1 \parallel \cdots \parallel \Gamma_n \vdash \mathcal{C} : R$. The cases for TC-Main, TC-Child, and TC-Link follow immediately. The cases for TC-New and TC-Par follow from the IH and structural congruence rules. □

It follows from Theorem 3.14 and Proposition 3.15 that any well-typed HGV configuration can be written as a forest of independent configurations in tree canonical form.

Progress and Deadlock Freedom. With tree canonical forms defined, we can now state a progress result. A thread is *blocked* on an endpoint x if it is ready to perform a communication action on x.

Definition 3.16 (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 $\textbf{send}\ (V, z)$, $\textbf{recv}\ z$, or $\textbf{wait}\ z$.

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 3.17 we obtain *open progress* for configurations with free runtime names.

Lemma 3.17 (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* $\mathcal{A}_i$ $(1 \leq i \leq n)$*,* $\mathsf{blocked}(\mathcal{A}_i, z)$ *for some* $z \in \{x_i\} \cup \{y_j \mid 1 \leq j < i\} \cup \mathrm{fv}(\Psi)$
(2) *Either* N *is a value or* $\mathsf{blocked}(\phi N, z)$ *for some* $z \in \{y_i \mid 1 \leq i \leq n\} \cup \mathrm{fv}(\Psi)$

Proof. Open progress follows as a direct corollary of a slightly more verbose property which holds on HGV processes, proved by induction on the derivation of an inductive definition of tree canonical forms. See Appendix A for details. □

Closed configurations enjoy a stronger result: if a closed configuration cannot reduce, then each auxiliary thread must either be a value, or be blocked on its neighbouring endpoint.

Lemma 3.18 (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 3.17, 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 tree canonical forms, $\mathcal{A}_2$ must contain x_2 and by the typing rules cannot contain y_2, so the thread must be blocked on x_2. The argument extends to the remainder of the configuration. □

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 [CDM14].

Definition 3.19 (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.

Our main progress result states that a ground configuration can reduce, or is a value.

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

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

$$\textsc{TG-Connect}_2\ \frac{\Gamma_1, y : \overline{S} \vdash_{\mathsf{GV}} \mathcal{C} : R \quad \Gamma_2, x : S \vdash_{\mathsf{GV}} \mathcal{D} : R'}{\Gamma_1, \Gamma_2, \langle x, y\rangle : S^\sharp \vdash_{\mathsf{GV}} \mathcal{C} \parallel \mathcal{D} : R \sqcap R'}$$

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

FIGURE 5. GV, typing rules for configurations.

Theorem 3.20 (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 3.18, 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, $\mathrm{fv}(V) = \emptyset$. By definition, tree canonical form ensures that no cycles are present amongst threads, so 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 it enjoys strong normalisation. Unlike with preservation and progress, the addition of hypersequents does not substantially change the arguments from [LM15].

Theorem 3.21 (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 [LM15, Fow19]: $\longrightarrow_{\mathsf{M}}$ is deterministic, and due to linearity, any overlapping reductions are separate and may be performed in either order. □

Theorem 3.22 (Termination)**.** *If $\mathcal{G} \vdash \mathcal{C} : T$, there are no infinite sequences $\mathcal{C} \longrightarrow\longrightarrow \cdots$.*

Proof. As with GV [LM15, Fow19], 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. □

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 configurations.

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.

GV uses a pseudo-type $S^\sharp$ to type channels. Unlike endpoints, channels cannot appear in terms. Read bottom-up, rule TG-New types a name restriction $(\nu xy)\mathcal{C}$, 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, this enforces a tree process structure. The remaining typing rules are as in HGV.

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. Let $\mathcal{C} \parallel_{\langle x,y\rangle} \mathcal{D}$ denote two configurations $\mathcal{C}$ and $\mathcal{D}$ connected by a channel with endpoints x, y. 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. In fact, we can provide a stronger result: every well-typed open GV configuration is exactly 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 type 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* $\Gamma \vdash_{\mathsf{GV}} \mathcal{C} : R$*, 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 $\Gamma \vdash_{\mathsf{GV}} \mathcal{C} : T$ (see Appendix B). □

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:

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

The corresponding HGV derivation is:

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

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

Translating HGV to GV. As we saw in §2, unlike in HGV, equivalence in GV is not type-preserving. It follows that HGV types strictly more processes than GV. Let us revisit Lindley and Morris' example from §1 (adapted to use double-binders), where $\Gamma_1, \Gamma_2, \Gamma_3 \vdash_{\mathsf{GV}} (\nu xx')(\nu yy')(\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})) : R_1 \sqcap R_2 \sqcap R_3$ with $\Gamma_1, x : S \vdash_{\mathsf{GV}} \mathcal{C} : R_1$, $\Gamma_2, y : S' \vdash_{\mathsf{GV}} \mathcal{D} : R_2$, and $\Gamma_3, x' : \overline{S}, y' : \overline{S'} \vdash_{\mathsf{GV}} \mathcal{E} : R_3$.

The structurally-equivalent term $(\nu xx')(\nu yy')((\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E})$ is not typeable in GV, since we cannot split both channels over a single parallel composition:

$$\dfrac{\dfrac{\dfrac{\Gamma_1, \Gamma_2, x : S \nvdash_{\mathsf{GV}} \mathcal{C} \parallel \mathcal{D} : R_1 \sqcap R_2 \qquad \Gamma_3, x' : \overline{S}, \langle y, y'\rangle : S'^\sharp \nvdash_{\mathsf{GV}} \mathcal{E} : R_3}{\Gamma_1, \Gamma_2, \Gamma_3, \langle x, x'\rangle : S^\sharp, \langle y, y'\rangle : S'^\sharp \nvdash_{\mathsf{GV}} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} : R_1 \sqcap R_2 \sqcap R_3}}{\Gamma_1, \Gamma_2, \Gamma_3, \langle x, x'\rangle : S^\sharp \nvdash_{\mathsf{GV}} (\nu yy')((\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}) : R_1 \sqcap R_2 \sqcap R_3}}{\Gamma_1, \Gamma_2, \Gamma_3 \nvdash_{\mathsf{GV}} (\nu xx')(\nu yy')((\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}) : R_1 \sqcap R_2 \sqcap R_3}$$

However, we *can* type this process in HGV:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1, x : S \vdash \mathcal{C} : R_1 \qquad \Gamma_2, y : S' \vdash \mathcal{D} : R_2}{\Gamma_1, x : S \parallel \Gamma_2, y : S' \vdash \mathcal{C} \parallel \mathcal{D} : R_1 \sqcap R_2} \qquad \Gamma_3, x' : \overline{S}, y' : \overline{S'} \vdash \mathcal{E} : R_3}{\boxed{\Gamma_1, x : S \parallel \Gamma_2, y : S' \parallel \Gamma_3, x' : \overline{S}, y' : \overline{S'}} \vdash (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} : R_1 \sqcap R_2 \sqcap R_3}}{\Gamma_1, x : S \parallel \Gamma_2, \Gamma_3, x' : \overline{S} \vdash (\nu yy')((\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}) : R_1 \sqcap R_2 \sqcap R_3}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash (\nu xx')(\nu yy')((\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}) : R_1 \sqcap R_2 \sqcap R_3}$$

Note in particular the shaded hyper-environment, which includes hyper-environment separators to separate endpoints x and x', as well as y and y'. It follows that, unlike in GV, *both* channels can be split over the same parallel composition. Similarly, the hyper-environment separator allows $\mathcal{C}$ and $\mathcal{D}$ to be composed *without* sharing any channels.

Although HGV types more processes, 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.

Proposition 4.5. *Suppose* $\Gamma \vdash \mathcal{C} : R$ *where* $\mathcal{C}$ *is in tree canonical form. Then,* $\Gamma \vdash_{\mathsf{GV}} \mathcal{C} : R$.

Proof. By induction on the derivation of $\Gamma \vdash \mathcal{C} : R$, making use of an inductive definition of tree canonical forms. See Appendix B for details. □

Remark 4.6. 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.

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

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

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

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

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

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

Figure 6. HCP, duality and typing rules for processes.

Corollary 4.7. *If* $\Gamma \vdash \mathcal{C} : R$*, then there exists some* $\mathcal{D}$ *such that* $\mathcal{C} \equiv \mathcal{D}$ *and* $\Gamma \vdash_{\mathsf{GV}} \mathcal{D} : R$*.*

5. Relation between HGV and HCP

In this section, we explore two translations, from HGV to HCP and from HCP to HGV, together with their operational correspondence results.

Hypersequent CP. HCP [MP18, KMP19b] 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 ($\otimes$, $\parr$) and units ($\mathbf{1}$, $\perp$) and the additive operators ($\oplus$, $\&$) and units ($\mathbf{0}$, $\top$).

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

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.

Processes (P, Q) are a variant of the π-calculus with forwarding [San96, Bor98], bound output [San96], and double binders [Vas12]. The syntax of processes is given by the typing rules (Figure 6), which are standard for HCP [MP18, KMP19b]: $x \leftrightarrow^{A} y$ forwards messages between x and y; $(\nu x y)P$ 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

Action rules

$$\textsc{Act-Pref}\quad \pi.P \xrightarrow{\pi} P \qquad \textsc{Act-Link}_1\quad x{\leftrightarrow}y \xrightarrow{x\leftrightarrow y} \mathbf{0} \qquad \textsc{Act-Link}_2\quad x{\leftrightarrow}y \xrightarrow{y\leftrightarrow x} \mathbf{0} \qquad \textsc{Act-Off-Inl}\quad x \triangleright \{\text{inl} : P; \text{inr} : Q\} \xrightarrow{x\triangleright\text{inl}} P$$

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

Communication Rules

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

$$\textsc{Bet-Inl}\quad \frac{P \xrightarrow{x\triangleleft\text{inl} \| y\triangleright\text{inl}} P'}{(\nu xy)P \xrightarrow{\beta} (\nu xy)P'} \qquad \textsc{Bet-Inr}\quad \frac{P \xrightarrow{x\triangleleft\text{inr} \| y\triangleright\text{inr}} P'}{(\nu xy)P \xrightarrow{\beta} (\nu xy)P'}$$

Structural Rules

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

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

FIGURE 7. HCP, label transition semantics.

$x[z].(z[].\mathbf{0} \parallel P))$ 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). Note that since we base our formal developments on an LTS semantics, structural congruence is not required for reduction.

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

$$(\nu xx')(\nu yy')P \equiv (\nu yy')(\nu xx')P \qquad (\nu xy)P \equiv (\nu yx)P$$

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

We define the labelled transition system for HCP as a small refinement of the LTS for the additive-multiplicative fragment of the πLL calculus introduced by Montesi and Peressotti [MP21], in turn inspired by their previous system *CT* [MP18]. The LTS is identical, save for the fact that we distinguish two types of internal actions. Action labels l represent the actions that a process can fire. Prefixes π are a convenient subset of action labels which can be written as prefixes to processes, *i.e.*, $\pi.P$. Transition labels ℓ include action labels and the parallel composition of two action labels, along with *internal* actions α, β, and τ. The LTS gives rise to two types of internal action: α represents only the evaluation of links as *renaming*, and β represents only *communication*. Labels τ arise only due to saturated transition (Definition 5.4) and are not produced by the rules in the LTS.

$$\begin{array}{llcl}
\text{Prefixes} & \pi & ::= & x[y] \mid x[] \mid x(y) \mid x() \mid x \triangleleft \text{inl} \mid x \triangleleft \text{inr} \\
\text{Action Labels} & l & ::= & \pi \mid x{\leftrightarrow}y \mid x \triangleright \text{inl} \mid x \triangleright \text{inr} \\
\text{Transition Labels} & \ell & ::= & l \mid l \parallel l' \mid \alpha \mid \beta
\end{array}$$

We let ℓ_x range over labels on x: $x{\leftrightarrow}y$, $x[y]$, $x[]$, *etc.* Labelled 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 $\xrightarrow{\ell^*}$ for the reflexive-transitive closure of $\xrightarrow{\ell}$. We write $\text{bn}(\ell)$ and $\text{fn}(\ell)$ for the bound and free names contained in ℓ, respectively. We write $\text{cn}(\ell)$ for all names in ℓ, *i.e.*, $\text{cn}(\ell) = \text{fn}(\ell) \cup \text{bn}(\ell)$.

Metatheory. Transitions preserve typeability. Since internal actions occur only under binders, they are typable under the same hyper-environment.

Theorem 5.1 (Type Preservation)**.** *Suppose $P \vdash \mathcal{G}$ and $P \xrightarrow{\ell} Q$.*

- *If ℓ is internal, then $Q \vdash \mathcal{G}$.*
- *If ℓ is not internal, then there exists some $\mathcal{H}$ such that $Q \vdash \mathcal{H}$.*

Proof. Following the approach of [KMP19a, MP18, MP21], type preservation is established by defining proof transformations on typing derivations of each reducing process. The only difference with respect to [MP18, MP21] arises due to our separate treatment of α and β actions, which does not materially impact the proof. □

Similarly, our LTS for HCP satisfies progress. Following [KMP19a, MP21], the key intermediate step is to note that for every type environment in a hyper-environment, there is some free name which can be acted upon. Again, the stratification of internal actions does not materially impact the proof.

Theorem 5.2 (Progress)**.** *If $P \vdash \mathcal{H}$ and $P \not\equiv \mathbf{0}$, then there exist some ℓ, Q such that $P \xrightarrow{\ell} Q$.*

Behavioural Theory. The behavioural theory for HCP follows Kokke *et al.* [KMP19a], except that we distinguish two subrelations of weak bisimilarity, following the subtypes of internal actions.

Definition 5.3 (Strong bisimulation and 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 5.4 (Saturated transition)**.** The *$\mathcal{L}$-saturated transition relation, for $\mathcal{L} \subseteq \{\alpha, \beta\}$, is the smallest relation $\Longrightarrow_{\mathcal{L}}$ closed under the following rules, with saturated transition labels ℓ ranging over transition labels and the distinguished label τ:*

$$\frac{}{P \overset{\tau}{\Longrightarrow}_{\mathcal{L}} P} \qquad \frac{P \overset{\tau}{\Longrightarrow}_{\mathcal{L}} P' \xrightarrow{\ell} Q' \overset{\tau}{\Longrightarrow}_{\mathcal{L}} Q \qquad \ell \in \mathcal{L}}{P \overset{\tau}{\Longrightarrow}_{\mathcal{L}} Q} \qquad \frac{P \overset{\tau}{\Longrightarrow}_{\mathcal{L}} P' \xrightarrow{\ell} Q' \overset{\tau}{\Longrightarrow}_{\mathcal{L}} Q \qquad \ell \notin \mathcal{L}}{P \overset{\ell}{\Longrightarrow}_{\mathcal{L}} Q}$$

We write $\Longrightarrow_{\ell}$ *as shorthand for* $\Longrightarrow_{\{\ell\}}$*, and we write* $\Longrightarrow$ *as shorthand for* $\Longrightarrow_{\{\alpha,\beta\}}$.

Definition 5.5 (Weak bisimulation and weak bisimilarity)**.** A symmetric relation $\mathcal{R}$ on processes is an $\mathcal{L}$-bisimulation, for $\mathcal{L} \subseteq \{\alpha, \beta\}$, if $P \mathrel{\mathcal{R}} Q$ implies that if $P \overset{\ell'}{\Longrightarrow}_{\mathcal{L}} P'$, then $Q \overset{\ell'}{\Longrightarrow}_{\mathcal{L}} Q'$ for some Q' such that $P' \mathrel{\mathcal{R}} Q'$. The $\mathcal{L}$-bisimilarity relation is the largest relation $\approx_{\mathcal{L}}$ that is an $\mathcal{L}$-bisimulation. We write $\approx$ as shorthand for $\approx_{\{\alpha,\beta\}}$.

Lemma 5.6. *Structural congruence, strong bisimilarity and the various forms of weak bisimilarity are related as follows:*

$$\equiv \subset \sim \qquad\qquad \sim \subset \approx \qquad\qquad \sim \subset \approx_{\alpha} \qquad\qquad \sim \subset \approx_{\beta}$$

Differences with previous version. The LTS in Figure 7 is similar to that in the previous version of this work [FKD+21], with the exception that we have removed the rules Tau-Alp and Tau-Bet:

$$\cancel{\dfrac{P \xrightarrow{\alpha} P'}{P \xrightarrow{\tau} P'}}\;\textsc{Tau-Alp} \qquad\qquad \cancel{\dfrac{P \xrightarrow{\beta} P'}{P \xrightarrow{\tau} P'}}\;\textsc{Tau-Bet}$$

To see why these rules are problematic, consider processes $P = (\nu xy)(z{\leftrightarrow}x \parallel y[].\mathbf{0})$ and $Q = z[].\mathbf{0}$. Following Definition 5.5, P and Q are α-bisimilar, as P only has the α-transition $P \xrightarrow{\alpha} Q$ and Q has no transitions. In the previous version, Tau-Alp gave P the derived τ-transition $P \xrightarrow{\tau} Q$, which meant that $P \not\approx_{\alpha} Q$, as $Q \overset{\tau}{\not\Longrightarrow} Q$. Therefore Tau-Alp collapses $\approx_{\alpha}$ to $\sim$ and Tau-Bet collapses $\approx_{\beta}$ to $\sim$.

The solution we adopted was to remove Tau-Alp and Tau-Bet from the label transition relation $\longrightarrow$, and instead lift α- and β-transitions to τ-transitions in the definition of saturated transition[1].

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 [LPT03] variant of HGV, which makes control flow explicit; and (2) a translation from HGV$*$ to HCP. In so doing, we can concentrate on the essence of the translations as opposed to concerning ourselves with administrative reductions.

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

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

[1] We thank Marco Peressotti for notifying us of the error and suggesting the fix.

Remark 5.7. Fine-grain call-by-value λ-calculi typically include an explicit $\textbf{return}\,V$ construct to embed values into the term language. As there is no difference between the shapes of the value and term typing judgements, we allow ourselves to embed values directly for simplicity.

We can *naïvely* translate HGV to HGV∗ ($(\!|\cdot|\!)$) by let-binding each subterm in a value position, *e.g.*, $(\!|\textbf{inl}\ M|\!) = \textbf{let}\ z = (\!|M|\!)\ \textbf{in}\ \textbf{inl}\ z$.

Definition 5.8 (Naïve translation of HGV to HGV∗)**.**

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

Standard techniques can be used to avoid administrative redexes [Plo75, DMN07]. We give a full definition of HGV∗ in Appendix C.

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, $\llbracket\cdot\rrbracket$ and $\lceil\!\lceil\cdot\rceil\!\rceil$, such that $\llbracket T\rrbracket = \lceil\!\lceil T\rceil\!\rceil^{\perp}$. We extend these translations pointwise to type environments and hyper-environments. We define translations on configurations ($\llbracket\cdot\rrbracket^{\mathsf{c}}_r$), terms ($\llbracket\cdot\rrbracket^{\mathsf{m}}_r$) and values ($\llbracket\cdot\rrbracket^{\mathsf{v}}_r$), where r is a fresh name denoting a distinguished output channel.

We translate an HGV sequent $\mathcal{G} \parallel \Gamma \vdash \mathcal{C} : T$ as $\llbracket\mathcal{C}\rrbracket^{\mathsf{c}}_r \vdash \llbracket\mathcal{G}\rrbracket \parallel \llbracket\Gamma\rrbracket, r : \llbracket T\rrbracket^{\perp}$, where Γ is the type environment corresponding to the main thread. The translation of computations includes synchronisation action in order to faithfully simulate a call-by-value reduction strategy. The (term) translation of a value $\llbracket V\rrbracket^{\mathsf{m}}_r$ immediately pings the output channel r to announce that it is a value. The translation of a let-binding $\llbracket\textbf{let}\ w = M\ \textbf{in}\ N\rrbracket^{\mathsf{m}}_r$ first evaluates M to a value, which then pings the internal channel x/x' and unblocks the continuation $x.\llbracket N\rrbracket^{\mathsf{m}}_r$. The translations of main and child threads each make use of an internal result channel. The translation of a child thread consumes the yielded unit endpoint once the child thread has terminated. The translation of the main thread forwards the result value along the external output channel once the main thread has terminated.

There are two changes with respect to the translation of our earlier paper [FKD+21]. First, in the earlier work the translation of the main thread output directly to the external output channel instead of forwarding via an intermediary as in the current translation. This change is purely aesthetic. Second, in the earlier work the translation of **fork** was not sufficiently concurrent. Correspondingly there was an error in the case of the operational correspondence proof which is fixed in the current paper.

Lemma 5.9 (Type Preservation)**.**

(1) *If* $\Gamma \vdash V : T$, *then* $\llbracket V\rrbracket^{\mathsf{v}}_r \vdash \llbracket\Gamma\rrbracket, r : \llbracket T\rrbracket^{\perp}$.
(2) *If* $\Gamma \vdash M : T$, *then* $\llbracket M\rrbracket^{\mathsf{m}}_r \vdash \llbracket\Gamma\rrbracket, r : \mathbf{1} \otimes \llbracket T\rrbracket^{\perp}$.

Translation on types $\boxed{\llparenthesis T \rrparenthesis \text{ and } \llbracket T \rrbracket}$

$$
\begin{array}{llll}
\llparenthesis !T.S \rrparenthesis = \llparenthesis T \rrparenthesis^{\perp} \otimes \llparenthesis S \rrparenthesis & \llparenthesis \mathbf{end}_{!} \rrparenthesis = \mathbf{1} & \llparenthesis T \rrparenthesis = \llbracket T \rrbracket^{\perp}, \\
\llparenthesis ?T.S \rrparenthesis = \llparenthesis T \rrparenthesis^{\perp} \parr \llparenthesis S \rrparenthesis & \llparenthesis \mathbf{end}_{?} \rrparenthesis = \perp & \quad \text{if } T \text{ is not a session type}
\end{array}
$$

$$
\begin{array}{lll}
\llbracket T \times U \rrbracket = \llbracket T \rrbracket \otimes \llbracket U \rrbracket & \llbracket \mathbf{1} \rrbracket = \mathbf{1} & \llbracket T \multimap U \rrbracket = \llbracket T \rrbracket^{\perp} \parr (\mathbf{1} \otimes \llbracket U \rrbracket) \\
\llbracket T + U \rrbracket = \llbracket T \rrbracket \oplus \llbracket U \rrbracket & \llbracket \mathbf{0} \rrbracket = \mathbf{0} & \llbracket S \rrbracket = \llparenthesis S \rrparenthesis^{\perp}
\end{array}
$$

Translation on configurations, terms, and values $\boxed{\llbracket \mathcal{C} \rrbracket^{\mathsf{c}}_r,\ \llbracket M \rrbracket^{\mathsf{m}}_r, \text{ and } \llbracket V \rrbracket^{\mathsf{v}}_r}$

$$
\begin{array}{lll}
\llbracket \circ\, M \rrbracket^{\mathsf{c}}_r = (\nu y y')(\llbracket M \rrbracket^{\mathsf{m}}_y \parallel y'.y'[].\mathbf{0}) & \llbracket (\nu x x')\mathcal{C} \rrbracket^{\mathsf{c}}_r = (\nu x x')\llbracket \mathcal{C} \rrbracket^{\mathsf{c}}_r & \llbracket x \overset{z}{\leftrightarrow} y \rrbracket^{\mathsf{c}}_r = \bar{z}.z().x \leftrightarrow y \\
\llbracket \bullet\, M \rrbracket^{\mathsf{c}}_r = (\nu y y')(\llbracket M \rrbracket^{\mathsf{m}}_y \parallel y'.y' \leftrightarrow r) & \llbracket \mathcal{C} \parallel \mathcal{D} \rrbracket^{\mathsf{c}}_r = \llbracket \mathcal{C} \rrbracket^{\mathsf{c}}_r \parallel \llbracket \mathcal{D} \rrbracket^{\mathsf{c}}_r &
\end{array}
$$

$$
\begin{array}{lll}
\llbracket x \rrbracket^{\mathsf{v}}_r = r \leftrightarrow x & \llbracket () \rrbracket^{\mathsf{v}}_r = r[].\mathbf{0} & \llbracket \mathbf{inl}\ V \rrbracket^{\mathsf{v}}_r = r \triangleleft \mathrm{inl}.\llbracket V \rrbracket^{\mathsf{v}}_r \\
\llbracket \lambda x.M \rrbracket^{\mathsf{v}}_r = r(x).\llbracket M \rrbracket^{\mathsf{m}}_r & \llbracket (V, W) \rrbracket^{\mathsf{v}}_r = r[x].(\llbracket V \rrbracket^{\mathsf{v}}_x \parallel \llbracket W \rrbracket^{\mathsf{v}}_r) & \llbracket \mathbf{inr}\ V \rrbracket^{\mathsf{v}}_r = r \triangleleft \mathrm{inr}.\llbracket V \rrbracket^{\mathsf{v}}_r
\end{array}
$$

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

$$
\begin{array}{ll}
\llbracket \mathbf{link} \rrbracket^{\mathsf{v}}_r & = r(y).y(x).\bar{r}.r().x \leftrightarrow y \\
\llbracket \mathbf{fork} \rrbracket^{\mathsf{v}}_r & = (\nu y y')(r(x).y\langle x \rangle.\bar{r}.r \leftrightarrow y \parallel y'(x).x\langle y' \rangle.x.x[].\mathbf{0}) \\
\llbracket \mathbf{send} \rrbracket^{\mathsf{v}}_r & = r(y).y(x).y\langle x \rangle.\bar{r}.r \leftrightarrow y \\
\llbracket \mathbf{recv} \rrbracket^{\mathsf{v}}_r & = r(x).x(y).\bar{r}.r\langle y \rangle.r \leftrightarrow x \\
\llbracket \mathbf{wait} \rrbracket^{\mathsf{v}}_r & = r(x).x().\bar{r}.r[].\mathbf{0}
\end{array}
$$

FIGURE 8. Translation from HGV$*$ to HCP.

(3) *If $\mathcal{G} \parallel \Gamma \vdash \mathcal{C} : T$, where Γ is the type environment for the main thread in $\mathcal{C}$, then $\llbracket \mathcal{C} \rrbracket^{\mathsf{c}}_r \vdash \llparenthesis \mathcal{G} \rrparenthesis \parallel \llparenthesis \Gamma \rrparenthesis, r : \llparenthesis T \rrparenthesis^{\perp}$.*

Lemma 5.10 (Substitution)**.** *If M is a well-typed term with $w \in \mathrm{fv}(M)$, and V is a well-typed value, then $(\nu w w')(\llbracket M \rrbracket^{\mathsf{m}}_r \parallel \llbracket V \rrbracket^{\mathsf{v}}_{w'}) \approx_{\alpha} \llbracket M\{V/w\} \rrbracket^{\mathsf{m}}_r$.*

Theorem 5.11 (Operational Correspondence)**.** *Suppose $\mathcal{C}$ is a well-typed configuration.*

(1) *(Preservation of reductions) If $\mathcal{C} \longrightarrow \mathcal{C}'$, then there exists a P such that $\llbracket \mathcal{C} \rrbracket^{\mathsf{c}}_r \overset{\beta^{+}}{\Longrightarrow}_{\alpha} P$ and $P \approx_{\alpha} \llbracket \mathcal{C}' \rrbracket^{\mathsf{c}}_r$; and*
(2) *(Reflection of transitions)*
   - *if $\llbracket \mathcal{C} \rrbracket^{\mathsf{c}}_r \xrightarrow{\alpha} P$, then $P \approx_{\alpha} \llbracket \mathcal{C} \rrbracket^{\mathsf{c}}_r$; and*
   - *if $\llbracket \mathcal{C} \rrbracket^{\mathsf{c}}_r \xrightarrow{\beta} P$, then there exists a $\mathcal{C}'$ and a P' such that $\mathcal{C} \longrightarrow \mathcal{C}'$ and $P \overset{\beta^{*}}{\Longrightarrow}_{\alpha} P'$ and $P' \approx_{\alpha} \llbracket \mathcal{C}' \rrbracket^{\mathsf{c}}_r$. Furthermore, $\mathcal{C}'$ is unique up to structural congruence.*

The proof is in Appendix C. One might strive for a tighter operational correspondence here, but our current translation generates multiple administrative β-transitions. The only term reduction that translates to multiple β-transitions is the one for let-bindings. This is because we choose to encode synchronisation using two β-transitions. We could adjust the accounting here by treating synchronisation as a single β-transition or its own special kind of administrative transition. Many more administrative reductions arise from the configuration translation. These are due to a combination of synchronisations and also the fact that we use constants along with pairs and application for our communication primitives instead of building-in fully-applied communication primitives.

Translating HCP to HGV. We cannot translate HCP processes to HGV terms directly: HGV's term language only supports **fork** (see Section 7 for further discussion), so there is no way to translate an individual name restriction or parallel composition. However, we can still translate HCP into HGV via the composition of known translations.

HCP into CP: We must first reunite each parallel composition with its corresponding name restriction, *i.e.*, translate to CP using the *disentanglement* translation shown by Kokke *et al.* [KMP19b, Lemma 4.7]. The result is a collection of independent CP processes.

CP into GV: Next, we can translate each CP process into a GV configuration using (a variant of) Lindley and Morris' translation [LM15, Figure 8].

GV into HGV: Finally, we can use our embedding of GV into HGV (Theorem 4.3) to obtain a collection of well-typed HGV configurations, which can be composed using TC-Par to result in a single well-typed HGV configuration.

The translation from HCP into CP and the embedding of GV into HGV preserve and reflect reduction. However, as previously mentioned, 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 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. 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 [LM15, Fow19]. Other extensions to GV such as shared channels [LM15], polymorphism [LM17], and recursive session types [LM16] adapt to HGV almost unchanged.

From trees to forests. 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.

The following TC-Mix rule 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 Mix rule [Gir87]; in the concurrent setting, Mix can be interpreted as concurrency *without* communication [LM15, ALM16]. TC-Mix admits a much simpler treatment of **link** and provides a crucial ingredient for handling exceptional behaviour.

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

Atkey *et al.* [ALM16] show that conflating the $\mathbf{1}$ and $\bot$ 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**.

$$\mathbf{fork} : (S \multimap \mathbf{1}) \multimap \overline{S} \qquad \mathbf{close} : \mathbf{end} \multimap \mathbf{1} \qquad \textsc{TC-Child}\ \frac{\Gamma \vdash M : \mathbf{1}}{\Gamma \vdash \circ M : \circ}$$

$$\textsc{E-Close}\quad (\nu xy)(E[\mathbf{close}\ x] \parallel E'[\mathbf{close}\ y]) \longrightarrow E[()] \parallel E'[()]$$

Given TC-Mix, we might expect a term-level construct $\mathbf{spawn} : (\mathbf{1} \multimap \mathbf{1}) \multimap \mathbf{1}$ 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:

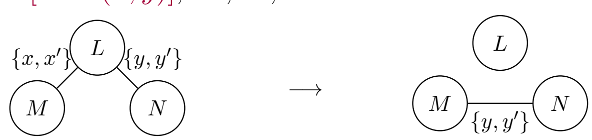
$$\cfrac{\cfrac{\cfrac{\cdot \vdash M : \mathbf{1}}{\cdot \vdash \circ M : \circ}\ \textsc{TC-Child} \qquad \cfrac{\cdot \vdash M : \mathbf{1}}{\cdot \vdash \bullet() : \mathbf{1}}\ \textsc{TC-Main}}{\cdot \parallel \cdot \vdash \circ M \parallel \bullet() : \mathbf{1}}\ \textsc{TC-Par}}{\cdot \vdash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

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 3.3 and Theorem 3.20.

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.* [FLMD19].

A simpler link. The $\mathbf{link}\ (x, y)$ construct forwards messages from x to y and vice-versa. Consider threads $L = F[\mathbf{link}\ (x, y)]$, 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 [LM15], which breaks confluence; another is two-stage linking (Figure 4), which defers forwarding via a special link thread [LM16].

Lindley and Morris [LM15] implement **link** using the following rule (modified here to use a double-binder formulation):

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

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** preemptively inhibits reduction in the second thread, since the evaluation rule inserts a blocking **wait**. The resulting system does not satisfy the diamond property.

HGV uses the incarnation of **link** advocated by Lindley and Morris [LM16], 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.

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:

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

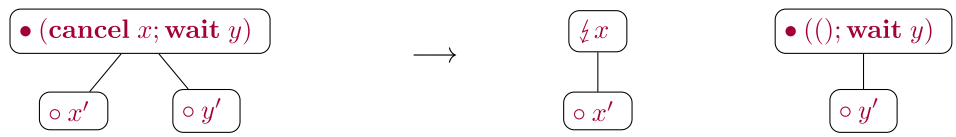
This formulation enables immediate substitution, maximimising concurrency. A variant of HGV replacing E-Reify-Link and E-Comm-Link with E-Link-Mix retains HGV's metatheory.

Exceptions. In order to support exceptions in the presence of linear endpoints [FLMD19, MV18] we must have a way of *cancelling* an endpoint. Mostrous and Vasconcelos [MV18] 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.* [FLMD19] extend these ideas to the functional setting, introducing Exceptional GV (EGV). EGV supports exceptional behaviour by adding three term-level constructs:

- 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 [BK01], 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

Cancellation generates a *zapper thread* ($\lightning x$) which severs a tree topology into a forest as in the following example.

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



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 typeable. We have described a *synchronous* version of EGV, but extending our treatment to asynchrony as in the work of [FLMD19] is a routine adaptation.

7. Can we separate fork?

Hyper-environments allow us to cleanly separate name restriction and parallel composition in process configurations. A natural follow-on question is whether we could use the same technique at the level of *terms* in order to split **fork** into separate constructs for creating a channel and spawning a process. As tantalising a prospect this is, we argue that the disadvantages outweigh the benefits.

Suppose we were to extend term typing to allow hyper-environments, $\mathcal{G} \vdash M : T$, and were to introduce terms $\textbf{let}\ \langle x, x' \rangle = \textbf{new in}\ M$ to create a channel and $\textbf{let}\ \langle\rangle = \textbf{spawn}\ M\ \textbf{in}\ N$ to spawn a thread, with the following typing rules:

TM-LetNew
$$\frac{\mathcal{G} \parallel \Gamma_1, x : S \parallel \Gamma_2, x' : \overline{S} \vdash M : T}{\mathcal{G} \parallel \Gamma_1, \Gamma_2 \vdash \textbf{let}\ \langle x, x' \rangle = \textbf{new in}\ M : T}$$

TM-LetSpawn
$$\frac{\mathcal{G} \vdash M : \textbf{end}_! \qquad \mathcal{H} \vdash N : T}{\mathcal{G} \parallel \mathcal{H} \vdash \textbf{let}\ \langle\rangle = \textbf{spawn}\ M\ \textbf{in}\ N : T}$$

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 $\textbf{fork}\ M$ as $\textbf{let}\ \langle x, x' \rangle = \textbf{new in let}\ \langle\rangle = \textbf{spawn}\ (M\ x)\ \textbf{in}\ x'$.

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 [MP18, KMP19b]; but (2) if we allow hyper-environments in *any logical rules*, we must use a semantics which allows the corresponding actions to be delayed [KMP19a]. 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. Therefore, we expect the positions where hypersequents can safely occur to follow from the structure of evaluation contexts and whether any blocking term perform a communication action.

Regardless of our choice, we would be left with restrictions on the syntax of terms that 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 (7.1) may be well-typed, but (7.2) is always ill-typed:

$$\textbf{let } y = 1 \textbf{ in let } \langle x, x' \rangle = \textbf{new in let } \langle\rangle = \textbf{spawn } M \textbf{ in } N \tag{7.1}$$

$$\textbf{let } \langle x, x' \rangle = \textbf{new in let } y = 1 \textbf{ in let } \langle\rangle = \textbf{spawn } M \textbf{ in } N \tag{7.2}$$

Note that $\textbf{let } \langle x, x' \rangle = \textbf{new in } M$ 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 types.

8. Related work

Session Types and Functional Languages. Session types were originally introduced in the context of process calculi [Hon93, THK94, HVK98], however they have been vastly integrated also in functional calculi, a line of work initiated by Gay and collaborators [VRG04, VGR06, GV10]. This family of calculi builds session types directly into a lambda calculus. Toninho *et al.* [TCP13] 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 [NT04, PT08, SE08, IYA10, OY16, KD21a]. A detailed survey is given by Orchard and Yoshida [OY17].

Propositions as Sessions. When Girard introduced linear logic [Gir87] he suggested a connection with concurrency. Abramsky [Abr94] and Bellin and Scott [BS94] 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 [CP10] 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 [Wad14], Wadler adapts πDILL to classical linear logic. Aschieri and Genco [AG20] 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 [Kob06, Pad14] and adopted by Dardha and Gay [DG18] in Priority CP (PCP), and Kokke and Dardha in Priority GV (PGV) [KD21b]. Dezani *et al.* [DCdY07] and Vieira and Vasconcelos [VV13] use a partial order on channels to guarantee deadlock freedom, following Kobayashi's work [Kob06]. Later on Dezani *et al.* [DCMYD06] guarantee progress by allowing only one active session at a time. Carbone *et al.* [CDM14] use catalysers to show that progress is a compositional form of lock freedom for standard typed π calculus. The authors describe how this technique can be used for session typed

π-calculus by using the the encoding of session types to linear types [DGS17, Dar14, Dar16]. Dardha and Perez [DP22] compare the different calculi and techniques for deadlock freedom using CP and CLL as a yardstick and showing that the class of processes in CP is strictly included in the class of processes typed by Kobayashi [Kob06].

Graph-theoretic Approaches. Carbone and Debois [CD10] define a graph-theoretic approach for a session typed π-calculus. They define an explicit dependency graph defined inductively on the structure of a process, in contrast to our approach of inducing a graph on type environments given a co-name set. They ensure progress for processes with acyclic graphs using a *catalyser*, which provides a missing counterpart to a process. Jacobs *et al*. [JBK22a] also define a graph-theoretic approach to deadlock freedom, but differently from Carbone and Debois, their work is based on separation logic. A line of work on many-writer, single-reader process calculi [Pad18, dP18] uses explicit dependency graphs to both ensure resource separation and guarantee deadlock freedom, however it is not immediate how to apply this approach to functional calculi.

9. 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 apply hypersequents to *multiparty* versions of CP [CLM$^+$16] and GV [JBK22b] to exhibit a similarly strong operational correspondence.

Acknowledgements

We are deeply grateful to Marco Peressotti for discovering an error in our presentation of the LTS for HCP in the original conference paper, and for suggesting a fix via the definition of saturated transitions. We thank the CONCUR and LMCS reviewers for their helpful comments and suggestions. This work was supported by EPSRC grants EP/K034413/1 (ABCD), EP/T014628/1 (STARDUST), EP/L01503X/1 (CDT PPar), ERC Consolidator Grant 682315 (Skye), UKRI Future Leaders Fellowship MR/T043830/1 (EHOP), a UK Government ISCF Metrology Fellowship grant, EU HORIZON 2020 MSCA RISE project 778233 (BehAPI), and NSF grant CCF-2044815.

References

[Abr94] Samson Abramsky. Proofs as processes. *Theoretical Computer Science*, 135(1):5–9, 1994. doi:10.1016/0304-3975(94)00103-0.

[AG20] 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. doi:10.1145/3371086.

[ALM16] 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. doi:10.1007/978-3-319-30936-1_2.

[Avr91] Arnon Avron. Hypersequents, logical consequence and intermediate logics for concurrency. *Ann. Math. Artif. Intell.*, 4:225–248, 1991. doi:10.1007/BF01531058.

[BK01] Nick Benton and Andrew Kennedy. Exceptional syntax. *J. Funct. Program.*, 11(4):395–410, 2001. doi:10.1017/S0956796801004099.

[Bor98] 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.

[BS94] Gianluigi Bellin and Philip J. Scott. On the pi-calculus and linear logic. *Theoretical Computer Science*, 135(1):11–65, 1994. doi:10.1016/0304-3975(94)00104-9.

[CD10] Marco Carbone and Søren Debois. A graphical approach to progress for structured communication in web services. In *Proc. of ICE*, volume 38 of *EPTCS*, pages 13–27, 2010. doi:10.4204/EPTCS.38.4.

[CDM14] 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.

[CLM+16] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *Proc. of CONCUR*, volume 59 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CONCUR.2016.33.

[CMS18] Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. *Distributed Comput.*, 31(1):51–67, 2018. doi:10.1007/s00446-017-0295-1.

[CP10] Luìs Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proc. of CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010. doi:10.1007/978-3-642-15375-4_16.

[Dar14] Ornela Dardha. Recursive session types revisited. In *Proc. of BEAT*, volume 162 of *Electron. Proc. in Theor. Comput. Sci.*, pages 27–34, 2014. doi:10.4204/EPTCS.162.4.

[Dar16] Ornela Dardha. *Type Systems for Distributed Programs: Components and Sessions*, volume 7 of *Atlantis Studies in Computing*. Springer / Atlantis Press, 2016. doi:10.2991/978-94-6239-204-5.

[DCdY07] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. On progress for structured communications. In *Proc. of TGC*, volume 4912 of *Lect. Notes Comput. Sci.*, pages 257–275. Springer, 2007. doi:10.1007/978-3-540-78663-4_18.

[DCMYD06] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *Proc. of ECOOP*, volume 4067 of *Lect. Notes Comput. Sci.*, pages 328–352. Springer, 2006. doi:10.1007/11785477_20.

[DG18] 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. doi:10.1007/978-3-319-89366-2_5.

[DGS17] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017. doi:10.1016/j.ic.2017.06.002.

[DMN07] Olivier Danvy, Kevin Millikin, and Lasse R. Nielsen. On one-pass CPS transformations. *J. Funct. Program.*, 17(6):793–812, 2007. doi:10.1017/S0956796807006387.

[dP18] Ugo de'Liguoro and Luca Padovani. Mailbox types for unordered interactions. In *Proc. of ECOOP*, volume 109 of *LIPIcs*, pages 15:1–15:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ECOOP.2018.15.

[DP22] Ornela Dardha and Jorge A. Pérez. Comparing type systems for deadlock freedom. *J. Log. Algebraic Methods Program.*, 124:100717, 2022. doi:10.1016/j.jlamp.2021.100717.

[FKD+21] Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. Separating sessions smoothly. In *Proc. of CONCUR*, volume 203 of *LIPIcs*, pages 36:1–36:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CONCUR.2021.36.

[FLMD19] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, 2019. doi:10.1145/3290341.

[Fow19] Simon Fowler. *Typed Concurrent Functional Programming with Channels, Actors, and Sessions*. PhD thesis, University of Edinburgh, 2019.

[Gir87] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi:10.1016/0304-3975(87)90045-4.

[GV10] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010. doi:10.1017/S0956796809990268.

[Hon93] Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

[HVK98] 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. doi:10.1007/BFb0053567.

[ITT+19] Atsushi Igarashi, Peter Thiemann, Yuya Tsuda, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *J. Funct. Program.*, 29:e17, 2019. doi:10.1017/S0956796819000169.

[IYA10] 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.

[JBK22a] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.*, 6(POPL):1–33, 2022.

[JBK22b] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proc. ACM Program. Lang.*, 6(ICFP):466–495, 2022.

[KD21a] Wen Kokke and Ornela Dardha. Deadlock-free session types in linear haskell. In *Proc. of Haskell*, pages 1–13. ACM, 2021. doi:10.1145/3471874.3472979.

[KD21b] Wen Kokke and Ornela Dardha. Prioritise the best variation. In *Proc. of FORTE*, volume 12719 of *Lecture Notes in Computer Science*, pages 100–119. Springer, 2021. doi:10.1007/978-3-030-78089-0_6.

[KMP19a] Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Better late than never: A fully-abstract semantics for classical processes. *PACMPL*, 3(POPL), 2019. doi:10.1145/3290337.

[KMP19b] 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, *Linearity-TLLA*, volume 292 of *EPTCS*, pages 90–103. Open Publishing Association, 2019. doi:10.4204/EPTCS.292.5.

[Kob03] Naoki Kobayashi. Type systems for concurrent programs. In Bernhard K. Aichernig and Tom Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support: 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002. Revised Papers*, pages 439–453. Springer Berlin Heidelberg, 2003. doi:10.1007/978-3-540-40007-3_26.

[Kob06] Naoki Kobayashi. A new type system for deadlock-free processes. In *Proc. of CONCUR*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006. doi:10.1007/11817949_16.

[LM99] 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. doi:10.1007/3-540-46691-6_14.

[LM15] 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. doi:10.1007/978-3-662-46669-8_23.

[LM16] Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. *SIGPLAN Not.*, 51(9):434–447, 2016. doi:10.1145/3022670.2951921.

[LM17] 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.

[LPT03] Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Inf. Comput.*, 185(2):182–210, 2003. doi:10.1016/S0890-5401(03)00088-9.

[Mon13] Fabrizio Montesi. *Choreographic Programming*. PhD thesis, IT University of Copenhagen, 2013.

[MP18] Fabrizio Montesi and Marco Peressotti. Classical transitions. *CoRR*, abs/1803.01049, 2018. URL: https://arxiv.org/abs/1803.01049, arXiv:1803.01049.

[MP21] Fabrizio Montesi and Marco Peressotti. Linear logic, the π-calculus, and their metatheory: A recipe for proofs as processes. *CoRR*, abs/2106.11818, 2021. URL: https://arxiv.org/abs/2106.11818, arXiv:2106.11818.

[MV18] Dimitris Mostrous and Vasco T. Vasconcelos. Affine sessions. *Log. Methods Comput. Sci.*, 14(4), 2018. doi:10.1007/978-3-662-43376-8_8.

[NT04] 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.

[OY16] 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.

[OY17] Dominic Orchard and Nobuko Yoshida. Session types with linearity in Haskell. *Behavioural Types: from Theory to Tools*, page 219, 2017.

[Pad14] Luca Padovani. Deadlock and Lock Freedom in the Linear π-Calculus. In *Proc. of CSL-LICS*, pages 72:1–72:10. ACM, 2014. doi:10.1145/2603088.2603116.

[Pad18] Luca Padovani. Deadlock-free typestate-oriented programming. *Art Sci. Eng. Program.*, 2(3):15, 2018. doi:10.22152/programming-journal.org/2018/2/15.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975. doi:10.1016/0304-3975(75)90017-1.

[PT08] Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Proc. of Haskell*. ACM, 2008. doi:10.1145/1411286.1411290.

[Rey00] John C. Reynolds. The meaning of types—from intrinsic to extrinsic semantics. Technical Report RS-00-32, BRICS, 2000.

[San96] 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.

[SE08] Matthew Sackman and Susan Eisenbach. Session types in Haskell: Updating message passing for the 21st century. Unpublished manuscript, 2008.

[TCP13] 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. doi:10.1007/978-3-642-37036-6_20.

[THK94] 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. doi:10.1007/3-540-58184-7_118.

[TV20] Peter Thiemann and Vasco T. Vasconcelos. Label-dependent session types. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–29, 2020. doi:10.1145/3371135.

[Vas12] Vasco T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012. doi:10.1016/j.ic.2012.05.002.

[VGR06] Vasco Thudichum Vasconcelos, Simon J. Gay, and Antonio Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006. doi:10.1016/j.tcs.2006.06.028.

[VRG04] Vasco Vasconcelos, Antonio Ravara, and Simon J. Gay. Session types for functional multithreading. In *CONCUR*, volume 3170 of *LNCS*, pages 497–511. Springer, 2004. doi:10.1007/978-3-540-28644-8_32.

[VV13] Hugo Torres Vieira and Vasco Thudichum Vasconcelos. Typing progress in communication-centred systems. In *Proc. of COORDINATION*, volume 7890 of *Lect. Notes Comput. Sci.*, pages 236–250. Springer, 2013. doi:10.1007/978-3-662-43376-8_10.

[Wad14] Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, 2014. doi:10.1145/2398856.2364568.

Appendices

Appendix A. Omitted Proofs for Section 3: Hypersequent GV

In this Appendix we give full definitions and proofs for Section 3.

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

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

$$\textsc{T-Select-Inr}\ \frac{}{\cdot \vdash \mathbf{select\ inr} : S \oplus S' \multimap S'} \qquad \textsc{T-Offer}\ \frac{\Gamma \vdash L : S \mathbin{\&} S' \qquad \Delta, x : S \vdash M : T \qquad \Delta, y : S' \vdash N : T}{\Gamma, \Delta \vdash \mathbf{offer}\ L\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} : T}$$

$$\textsc{T-Offer-Absurd}\ \frac{\Gamma \vdash L : \&\{\}}{\Gamma, \Delta \vdash \mathbf{offer}\ L\ \{\} : T}$$

Figure 9. Derived rules for syntactic sugar

A.1. Derived typing rules for syntactic sugar.

The main body makes use of syntactic sugar, and encodings of branching and selection. Figure 9 shows the derived typing rules.

A.2. Preservation Proof.

Next, we detail the proof of preservation. We begin with the usual lemmas to manipulate evaluation contexts, and the usual substitution lemma.

Lemma A.1 (Subterm typeability)**.** *Suppose* $\mathbf{D}$ *is a derivation of* $\Gamma \vdash E[M] : T$*. Then, there exist* Γ_1, Γ_2 *such that* $\Gamma = \Gamma_1, \Gamma_2$*, 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 A.2 (Replacement, Evaluation Contexts)**.** *If:*

- **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 A.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 $\Gamma_1, x : U \vdash M : T$. ☐

Preservation of typing under term reduction is standard.

Lemma A.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 A.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 $\sqcap$. ☐

The first more major result is preservation of configuration typing under structural congruence.

Lemma A.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).

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

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

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).

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

$$\Longleftrightarrow$$

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

Subcase (2).

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

$$\Longleftrightarrow$$

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

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

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

$$\Longleftrightarrow$$

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

Case (SC-LinkComm).

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

Assumption:

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

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. ☐

The next result shows that configuration typeability is preserved under configuration reduction. Note that this lemma makes crucial use of Lemma A.6 due to E-Equiv.

Lemma A.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:

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

By Lemma A.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:

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

By Lemma A.2:

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

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:

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

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:

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

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

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

With the knowledge that $S = !T.S'$, we can refine our original derivation:

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

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

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

We can show:

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

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

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

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:

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

By Lemma A.1, we have that:

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

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

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

as required.

Case (E-Reify-Link).

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

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

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

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

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

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

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

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:

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

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

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

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 A.6, $\mathcal{G} \vdash \mathcal{C}' : R$.
By the IH, $\mathcal{G} \vdash \mathcal{D}' : R$.
By Lemma A.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 A.4. □

Theorem 3.3 (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 A.6 and A.7. □

A.3. **Progress.** 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 A.8 (Progress, Terms)**.** *If* $\Psi \vdash M : T$*, then either* M *is a value, or there exists some* N *such that* $M \longrightarrow_{\mathsf{M}} N$*, or* M *can be written* $E[N]$ *for some* $N \in \{\mathbf{fork}\ V, \mathbf{send}\ (V, W), \mathbf{recv}\ V, \mathbf{wait}\ V, \mathbf{link}\ (V, W)\}$.

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

Note that tree canonical forms can be defined inductively:

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

We assume the same requirement for configurations $\mathcal{F}$ as the non-inductive definition of tree canonical forms: i.e., that for a configuration $(\nu xy)(\mathcal{A} \parallel \mathcal{F})$, that $x \in \mathrm{fv}(\mathcal{A})$.

Lemma 3.17 follows as a direct corollary of a slightly more verbose property, which follows from the inductive definition of TCFs.

Definition A.9 (Open progress)**.** Suppose $\Psi \vdash \mathcal{F} : R$, where $\mathcal{F} \not\longrightarrow$. We say that $\mathcal{F}$ *satisfies open progress* if:

(1) $\mathcal{C} = (\nu x x')(\mathcal{A} \parallel \mathcal{F}')$, 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 $\mathcal{F}'$ satisfies open progress

(2) $\mathcal{F} = \phi M$, and either M is a value, or $\mathsf{blocked}(\phi M, x)$ for some $x \in \mathrm{fv}(\Psi)$.

Lemma A.10 (Open progress)**.** *If* $\Psi \vdash \mathcal{F} : R$ *and* $\mathcal{F} \not\longrightarrow$*, then* $\mathcal{F}$ *satisfies open progress.*

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

We show the case where $\mathcal{C} = (\nu x y)(\circ M \parallel \mathcal{F}')$; the cases for $\mathcal{A} = x \overset{z}{\leftrightarrow} x'$ and $\mathcal{C} = \bullet M$ follow similar reasoning.

Assumption:

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

In both cases, by the induction hypothesis, $\Psi_2, y : \overline{S} \vdash \mathcal{F}' : T$ satisfies open progress.

Subcase ($\mathcal{A} = \circ M$). By Lemma A.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 $z, z' \in \mathrm{fv}(\Psi, x : S)$ and could reduce by E-Reify-Link.

Otherwise, it must be the case that $\mathsf{blocked}(\circ M, z)$ for some $z \in \mathrm{fv}(\Psi_1, x : S)$.

Thus, $(\nu x y)(\circ M \parallel \mathcal{D})$ satisfies open progress, as required.

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

Appendix B. Omitted Proofs for section 4: Relation between HGV and GV

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:

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

Suppose $\Gamma = \langle x_1, x'_1 \rangle : S^\sharp_1, \ldots, \langle x_n, x'_n \rangle : S^\sharp_n$ (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 3.12 (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 3.12 (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. □

Proposition 4.5. *Suppose* $\Gamma \vdash \mathcal{C} : R$ *where* $\mathcal{C}$ *is in tree canonical form. Then,* $\Gamma \vdash_{\mathsf{GV}} \mathcal{C} : R$.

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:

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

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. □

Appendix C. Omitted Proofs for section 5: Relation between HGV and CP

C.1. Full definition of HGV∗.

Syntax.

Terms	L, M, N	$::=$	$V \mid \mathbf{let}\ x = M\ \mathbf{in}\ N \mid V\ W$
		$\mid$	$\mathbf{let}\ () = V\ \mathbf{in}\ M \mid \mathbf{let}\ (x, y) = V\ \mathbf{in}\ M$
		$\mid$	$\mathbf{absurd}\ V \mid \mathbf{case}\ V\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\}$
Values	V, W	$::=$	$x \mid K \mid \lambda x.M \mid () \mid (V, W) \mid \mathbf{inl}\ V \mid \mathbf{inr}\ V$
Evaluation contexts	E	$::=$	$\square \mid \mathbf{let}\ x = E\ \mathbf{in}\ M$
Thread contexts	F	$::=$	$\phi\ E$

Typing rules for values $\boxed{\Gamma \vdash V : T}$

$$\frac{}{x : T \vdash x : T}\ \text{TV*-Var} \qquad \frac{}{\cdot \vdash K : T}\ \text{TV*-Const} \qquad \frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x.M : T \multimap U}\ \text{TV*-Lam} \qquad \frac{}{\cdot \vdash () : \mathbf{1}}\ \text{TV*-Unit}$$

$$\frac{\Gamma \vdash V : T \qquad \Delta \vdash W : U}{\Gamma, \Delta \vdash (V, W) : T \times U}\ \text{TV*-Pair} \qquad \frac{\Gamma \vdash V : \mathbf{0}}{\Gamma \vdash \mathbf{absurd}\ V : T}\ \text{TV*-Absurd} \qquad \frac{\Gamma \vdash V : T}{\Gamma \vdash \mathbf{inl}\ V : T + U}\ \text{TV*-Inl} \qquad \frac{\Gamma \vdash V : U}{\Gamma \vdash \mathbf{inr}\ V : T + U}\ \text{TV*-Inr}$$

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

$$\frac{\Gamma \vdash V : T \multimap U \qquad \Delta \vdash W : T}{\Gamma, \Delta \vdash V\ W : U}\ \text{TM*-App} \qquad \frac{\Gamma \vdash M : T \qquad \Delta, x : T \vdash N : U}{\Gamma, \Delta \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : U}\ \text{TM*-Let}$$

$$\frac{\Gamma \vdash V : \mathbf{1} \qquad \Delta \vdash M : T}{\Gamma, \Delta \vdash \mathbf{let}\ () = V\ \mathbf{in}\ M : T}\ \text{TM*-LetUnit} \qquad \frac{\Gamma \vdash V : T \times T' \qquad \Delta, x : T, y : T' \vdash M : U}{\Gamma, \Delta \vdash \mathbf{let}\ (x, y) = V\ \mathbf{in}\ M : U}\ \text{TM*-LetPair}$$

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

The typing of constants is the same as for HGV.

Operational Semantics. The operational semantics for HGV$*$ is the same as for HGV (Figure 4), with the addition of the following explicit rule for **let**:

$$\text{E-Let} \quad \mathbf{let}\ x = V\ \mathbf{in}\ M \quad \longrightarrow \quad M\{V/x\}$$

Similarly, HGV$*$ directly inherits HGV's runtime typing.

C.2. **Translating HGV$*$ to HCP.** The translation is guaranteed to have only internal (i.e., α or β) transitions and transitions on the dedicated output channel. More specifically:

Lemma C.1.

- *If* $[\![C]\!]^{\mathsf{c}}_r \xrightarrow{\ell}$, *then* $\ell \in \{\alpha, \beta\}$ *or* $\ell = \ell_r$.
- *If* $[\![M]\!]^{\mathsf{m}}_r \xrightarrow{\ell}$ *and* M *is a non-value, then* $\ell \in \{\alpha, \beta\}$.
- *If* V *is a value, then* $[\![V]\!]^{\mathsf{m}}_r \xrightarrow{\ell_r}$.
- *If* $[\![V]\!]^{\mathsf{v}}_r \xrightarrow{\ell}$ *then* $\ell \in \{\alpha, \beta\}$.

Proof. By induction on the structure of M. □

We do not use the above lemma directly, but it is a useful sanity check.

Definition C.2 (process contexts)**.** A process context $P[\,]$ is a process with a single hole, denoted $\Box$. 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 C.3. A process P is *blocked on* x if it only has transitions $P \xrightarrow{\ell_x}$.

Lemma C.4. *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 ($\Box$).

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

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

$$\begin{array}{lll} & (\nu w w')((\nu xy)(P[z{\leftrightarrow}w]) \parallel Q) & \\ \sim & (\nu xy)(\nu w w')(P[z{\leftrightarrow}w] \parallel Q) & \text{(by Lemma 5.6)} \\ \approx_\alpha & (\nu xy)(P[Q\{z/w'\}]) & \text{(by Lemma 5.6 and IH)} \end{array}$$

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

$$\begin{array}{lll} & (\nu w w')(P[z{\leftrightarrow}w] \parallel R \parallel Q) & \\ \sim & (\nu w w')(P[z{\leftrightarrow}w] \parallel Q) \parallel R & \text{(by Lemma 5.6)} \\ \approx_\alpha & P[Q\{z/w'\}] \parallel R & \text{(by Lemma 5.6 and IH)} \end{array}$$

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

$$\begin{array}{lll} & (\nu w w')(R \parallel P[z{\leftrightarrow}w] \parallel Q) & \\ \sim & R \parallel (\nu w w')(P[z{\leftrightarrow}w] \parallel Q) & \text{(by Lemma 5.6)} \\ \approx_\alpha & R \parallel P[Q\{z/w'\}] & \text{(by Lemma 5.6 and IH)} \end{array}$$

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

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

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 $(\nu w w')(x \triangleright \{\mathrm{inl} : P[z{\leftrightarrow}w]; \mathrm{inr} : P'[z{\leftrightarrow}w]\} \parallel Q)$ has only two transitions,

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

and

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

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. $\Box$

Lemma 5.10 (Substitution)**.** *If M is a well-typed term with $w \in \mathrm{fv}(M)$, and V is a well-typed value, then $(\nu w w')(\llbracket M \rrbracket^{\mathsf{m}}_r \parallel \llbracket V \rrbracket^{\mathsf{v}}_{w'}) \approx_\alpha \llbracket M\{V/w\} \rrbracket^{\mathsf{m}}_r$.*

Proof. Immediately from Lemma C.4. $\Box$

Lemma 5.9 (Type Preservation)**.**

(1) *If* $\Gamma \vdash V : T$, *then* $[\![V]\!]^{\mathsf{v}}_r \vdash \llbracket\Gamma\rrbracket, r : \llbracket T\rrbracket^\perp$.
(2) *If* $\Gamma \vdash M : T$, *then* $[\![M]\!]^{\mathsf{m}}_r \vdash \llbracket\Gamma\rrbracket, r : \mathbf{1} \otimes \llbracket T\rrbracket^\perp$.
(3) *If* $\mathcal{G} \parallel \Gamma \vdash \mathcal{C} : T$, *where* Γ *is the type environment for the main thread in* $\mathcal{C}$, *then* $[\![\mathcal{C}]\!]^{\mathsf{c}}_r \vdash \llbracket\mathcal{G}\rrbracket \parallel \llbracket\Gamma\rrbracket, r : \llbracket T\rrbracket^\perp$.

Proof. Part 1.

- **Case** (x). We have $\overline{x : T \vdash x : T}$ and $[\![x]\!]^{\mathsf{v}}_r = r \leftrightarrow x$. We can derive:

$$\frac{}{x \leftrightarrow r \vdash x : \llbracket A\rrbracket, r : \llbracket A\rrbracket^\perp}$$

- **Case** (K). We have one case for each communication primitive.
  - Subcase **link**. We have $\mathbf{link} : S \times \overline{S} \multimap \mathbf{end}_!$, where

$$\begin{aligned}\llbracket S \times S^\perp \multimap \mathbf{end}_!\rrbracket^\perp &= \lceil\!\lceil S \times S^\perp \multimap \mathbf{end}_!\rceil\!\rceil \\ &= \lceil\!\lceil S \times S^\perp\rceil\!\rceil^\perp \parr (\mathbf{1} \otimes \lceil\!\lceil \mathbf{end}_!\rceil\!\rceil) \\ &= (\lceil\!\lceil S\rceil\!\rceil^\perp \parr \lceil\!\lceil S\rceil\!\rceil) \parr (\mathbf{1} \otimes \perp)\end{aligned}$$

  and $[\![\mathbf{link}]\!]^{\mathsf{v}}_r = r(y).y(x).\bar{r}.r().x \leftrightarrow y$. We can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x \leftrightarrow y \vdash x : \lceil\!\lceil S\rceil\!\rceil^\perp, y : \lceil\!\lceil S\rceil\!\rceil}}{r().x \leftrightarrow y \vdash x : \lceil\!\lceil S\rceil\!\rceil^\perp, y : \lceil\!\lceil S\rceil\!\rceil, r : \perp}}{\bar{r}.r().x \leftrightarrow y \vdash x : \lceil\!\lceil S\rceil\!\rceil^\perp, y : \lceil\!\lceil S\rceil\!\rceil, r : \mathbf{1} \otimes \perp}}{y(x).\bar{r}.r().x \leftrightarrow y \vdash y : \lceil\!\lceil S\rceil\!\rceil^\perp \parr \lceil\!\lceil S\rceil\!\rceil, r : \mathbf{1} \otimes \perp}}{r(y).y(x).\bar{r}.r().x \leftrightarrow y \vdash r : (\lceil\!\lceil S\rceil\!\rceil^\perp \parr \lceil\!\lceil S\rceil\!\rceil) \parr (\mathbf{1} \otimes \perp)}$$

  - Subcase **fork**: We have $\mathbf{fork} : (S \multimap \mathbf{end}_!) \multimap \overline{S}$ where

$$\begin{aligned}\llbracket (S \multimap \mathbf{end}_!) \multimap S^\perp\rrbracket^\perp &= \lceil\!\lceil (S \multimap \mathbf{end}_!) \multimap S^\perp\rceil\!\rceil \\ &= \lceil\!\lceil S \multimap \mathbf{end}_!\rceil\!\rceil^\perp \parr (\mathbf{1} \otimes \lceil\!\lceil S^\perp\rceil\!\rceil) \\ &= (\lceil\!\lceil S\rceil\!\rceil^\perp \parr (\mathbf{1} \otimes \lceil\!\lceil \mathbf{end}_!\rceil\!\rceil))^\perp \parr (\mathbf{1} \otimes \llbracket S\rrbracket) \\ &= (\llbracket S\rrbracket^\perp \otimes (\perp \parr \mathbf{1})) \parr (\mathbf{1} \otimes \llbracket S\rrbracket)\end{aligned}$$

  and $[\![\mathbf{fork}]\!]^{\mathsf{v}}_r = (\nu y y')(r(x).y\langle x\rangle.\bar{r}.r \leftrightarrow y \parallel y'(x).x\langle y'\rangle.x.x[].\mathbf{0})$. We derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{r \leftrightarrow y \vdash y : \llbracket S\rrbracket^\perp, r : \llbracket S\rrbracket}}{\bar{r}.r \leftrightarrow y \vdash y : \llbracket S\rrbracket^\perp, r : \mathbf{1} \otimes \llbracket S\rrbracket}}{y\langle x\rangle.\bar{r}.r \leftrightarrow y \vdash y : (\llbracket S\rrbracket \parr (\mathbf{1} \otimes \perp)) \otimes \llbracket S\rrbracket^\perp, x : (\llbracket S\rrbracket^\perp \otimes (\perp \parr \mathbf{1})), r : (\mathbf{1} \otimes \llbracket S\rrbracket)}}{(r(x).y\langle x\rangle.\bar{r}.r \leftrightarrow y) \vdash y : (\llbracket S\rrbracket \parr (\mathbf{1} \otimes \perp)) \otimes \llbracket S\rrbracket^\perp, r : (\llbracket S\rrbracket^\perp \otimes (\perp \parr \mathbf{1})) \parr (\mathbf{1} \otimes \llbracket S\rrbracket)} \quad \mathcal{D}}{(r(x).y\langle x\rangle.\bar{r}.r \leftrightarrow y \parallel y'(x).x\langle y'\rangle.x.x[].\mathbf{0}) \vdash y : T, r : (\llbracket S\rrbracket^\perp \otimes (\perp \parr \mathbf{1})) \parr (\mathbf{1} \otimes \llbracket S\rrbracket) \parallel y' : T^\perp}}{(\nu y y')(r(x).y\langle x\rangle.\bar{r}.r \leftrightarrow y \parallel y'(x).x\langle y'\rangle.x.x[].\mathbf{0}) \vdash r : (\llbracket S\rrbracket^\perp \otimes (\perp \parr \mathbf{1})) \parr (\mathbf{1} \otimes \llbracket S\rrbracket)}$$

  where

$$\begin{aligned}T &= (\llbracket S\rrbracket \parr (\mathbf{1} \otimes \perp)) \otimes \llbracket S\rrbracket^\perp \\ T^\perp &= (\llbracket S\rrbracket^\perp \otimes (\perp \parr \mathbf{1})) \parr \llbracket S\rrbracket\end{aligned}$$

and $\mathcal{D}$ is the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{0} \vdash \varnothing}}{x[].\mathbf{0} \vdash x : \mathbf{1}}}{x.x[].\mathbf{0} \vdash x : (\bot \parr \mathbf{1})}}{x\langle y' \rangle.x.x[].\mathbf{0} \vdash x : (\llbracket S \rrbracket^{\bot} \otimes (\bot \parr \mathbf{1})), y' : \llbracket S \rrbracket}}{y'(x).x\langle y' \rangle.x.x[].\mathbf{0} \vdash y' : (\llbracket S \rrbracket^{\bot} \otimes (\bot \parr \mathbf{1})) \parr \llbracket S \rrbracket}$$

– Subcase **send**: We have $\mathbf{send} : T \times !T.S \multimap S$ where

$$\begin{aligned} \llbracket T \times !T.S \multimap S \rrbracket^{\bot} &= \lceil\!\lceil T \times !T.S \multimap S \rceil\!\rceil \\ &= \lceil\!\lceil T \times !T.S \rceil\!\rceil^{\bot} \parr (1 \otimes \lceil\!\lceil S \rceil\!\rceil) \\ &= (\lceil\!\lceil T \rceil\!\rceil^{\bot} \parr \llbracket !T.S \rrbracket) \parr (1 \otimes \lceil\!\lceil S \rceil\!\rceil) \\ &= (\lceil\!\lceil T \rceil\!\rceil^{\bot} \parr (\lceil\!\lceil T \rceil\!\rceil \otimes \llbracket S \rrbracket)) \parr (1 \otimes \llbracket S \rrbracket^{\bot}) \end{aligned}$$

and $\llbracket \mathbf{send} \rrbracket^{\vee}_{r} = r(y).y(x).y\langle x \rangle.\bar{r}.r{\leftrightarrow}y$. We derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{r{\leftrightarrow}y \vdash y : \llbracket S \rrbracket, r : \llbracket S \rrbracket^{\bot}}}{\bar{r}.r{\leftrightarrow}y \vdash y : \llbracket S \rrbracket, r : (1 \otimes \llbracket S \rrbracket^{\bot})}}{y\langle x \rangle.\bar{r}.r{\leftrightarrow}y \vdash x : \lceil\!\lceil T \rceil\!\rceil^{\bot}, y : (\lceil\!\lceil T \rceil\!\rceil \otimes \llbracket S \rrbracket), r : (1 \otimes \llbracket S \rrbracket^{\bot})}}{y(x).y\langle x \rangle.\bar{r}.r{\leftrightarrow}y \vdash y : (\lceil\!\lceil T \rceil\!\rceil^{\bot} \parr (\lceil\!\lceil T \rceil\!\rceil \otimes \llbracket S \rrbracket)), r : (1 \otimes \llbracket S \rrbracket^{\bot})}}{r(y).y(x).y\langle x \rangle.\bar{r}.r{\leftrightarrow}y \vdash r : (\lceil\!\lceil T \rceil\!\rceil^{\bot} \parr (\lceil\!\lceil T \rceil\!\rceil \otimes \llbracket S \rrbracket)) \parr (1 \otimes \llbracket S \rrbracket^{\bot})}$$

– Subcase **recv**: We have $\mathbf{recv} : ?T.S \multimap T \times S$ where

$$\begin{aligned} \llbracket ?T.S \multimap T \times S \rrbracket^{\bot} &= \lceil\!\lceil ?T.S \multimap T \times S \rceil\!\rceil \\ &= \lceil\!\lceil ?T.S \rceil\!\rceil^{\bot} \parr (1 \otimes \lceil\!\lceil T \times S \rceil\!\rceil) \\ &= (\lceil\!\lceil T \rceil\!\rceil^{\bot} \parr \lceil\!\lceil S \rceil\!\rceil^{\bot}) \parr (1 \otimes (\lceil\!\lceil T \rceil\!\rceil \otimes \lceil\!\lceil S \rceil\!\rceil)) \end{aligned}$$

and $\llbracket \mathbf{recv} \rrbracket^{\vee}_{r} = r(x).x(y).\bar{r}.r\langle y \rangle.r{\leftrightarrow}x$. We derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{r{\leftrightarrow}x \vdash x : \lceil\!\lceil S \rceil\!\rceil^{\bot}, r : \lceil\!\lceil S \rceil\!\rceil}}{r\langle y \rangle.r{\leftrightarrow}x \vdash y : \lceil\!\lceil T \rceil\!\rceil^{\bot}, x : \lceil\!\lceil S \rceil\!\rceil^{\bot}, r : \lceil\!\lceil T \rceil\!\rceil \otimes \lceil\!\lceil S \rceil\!\rceil}}{\bar{r}.r\langle y \rangle.r{\leftrightarrow}x \vdash y : \lceil\!\lceil T \rceil\!\rceil^{\bot}, x : \lceil\!\lceil S \rceil\!\rceil^{\bot}, r : (1 \otimes (\lceil\!\lceil T \rceil\!\rceil \otimes \lceil\!\lceil S \rceil\!\rceil))}}{x(y).\bar{r}.r\langle y \rangle.r{\leftrightarrow}x \vdash x : (\lceil\!\lceil T \rceil\!\rceil^{\bot} \parr \lceil\!\lceil S \rceil\!\rceil^{\bot}), r : (1 \otimes (\lceil\!\lceil T \rceil\!\rceil \otimes \lceil\!\lceil S \rceil\!\rceil))}}{r(x).x(y).\bar{r}.r\langle y \rangle.r{\leftrightarrow}x \vdash r : (\lceil\!\lceil T \rceil\!\rceil^{\bot} \parr \lceil\!\lceil S \rceil\!\rceil^{\bot}) \parr (1 \otimes (\lceil\!\lceil T \rceil\!\rceil \otimes \lceil\!\lceil S \rceil\!\rceil))}$$

– Subcase **wait**: We have $\mathbf{wait} : \mathbf{end}_? \multimap \mathbf{1}$ where

$$\begin{aligned} \llbracket \mathbf{end}_? \multimap \mathbf{1} \rrbracket^{\bot} &= \lceil\!\lceil \mathbf{end}_? \multimap \mathbf{1} \rceil\!\rceil \\ &= \lceil\!\lceil \mathbf{end}_? \rceil\!\rceil^{\bot} \parr (\mathbf{1} \otimes \lceil\!\lceil \mathbf{1} \rceil\!\rceil) \\ &= \bot \parr (\mathbf{1} \otimes \mathbf{1}) \end{aligned}$$

and $\llbracket \textbf{wait} \rrbracket^{\mathsf{v}}_r = r(x).x().\bar{r}.r[].\mathbf{0}$. We derive

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{0} \vdash \varnothing}}{r[].\mathbf{0} \vdash r : \mathbf{1}}}{\bar{r}.r[].\mathbf{0} \vdash r : \mathbf{1} \otimes \mathbf{1}}}{x().\bar{r}.r[].\mathbf{0} \vdash x : \bot, r : \mathbf{1} \otimes \mathbf{1}}}{r(x).x().\bar{r}.r[].\mathbf{0} \vdash r : \bot \parr (\mathbf{1} \otimes \mathbf{1})}$$

- **Case** ($\lambda x.M$). We assume $\llbracket M \rrbracket^{\mathsf{m}}_r : \llfloor \Gamma \rrfloor, x : \llfloor T \rrfloor, r : \mathbf{1} \otimes \llfloor U \rrfloor^{\bot}$ and derive

$$\dfrac{\llbracket M \rrbracket^{\mathsf{m}}_r \vdash \llfloor \Gamma \rrfloor, x : \llfloor T \rrfloor, r : \mathbf{1} \otimes \llceil U \rrceil}{r(x).\llbracket M \rrbracket^{\mathsf{m}}_r \vdash \llfloor \Gamma \rrfloor, r : \llfloor T \rrfloor \parr (\mathbf{1} \otimes \llceil U \rrceil)}$$

- **Case** ($()$). We derive:

$$\dfrac{\mathbf{0} \vdash \varnothing}{x[].\mathbf{0} \vdash x : \mathbf{1}}$$

- **Case** ($\textbf{inl}\ W$). We assume $\llbracket W \rrbracket^{\mathsf{v}}_r : \llfloor \Gamma \rrfloor, r : \llfloor T \rrfloor^{\bot}$ and derive

$$\dfrac{\llbracket W \rrbracket^{\mathsf{v}}_r \vdash \llfloor \Gamma \rrfloor, r : \llfloor T \rrfloor^{\bot}}{r \triangleleft \mathrm{inl}.\llbracket W \rrbracket^{\mathsf{v}}_r \vdash \llfloor \Gamma \rrfloor, r : \llceil T \rrceil \oplus \llceil U \rrceil}$$

- **Case** ((V, W)). We assume $\llbracket V \rrbracket^{\mathsf{v}}_x : \llfloor \Gamma \rrfloor, x : \llfloor T \rrfloor^{\bot}$, $\llbracket W \rrbracket^{\mathsf{v}}_r : \llfloor \Delta \rrfloor, r : \llfloor U \rrfloor^{\bot}$, and derive

$$\dfrac{\dfrac{\llbracket V \rrbracket^{\mathsf{v}}_x \vdash \llfloor \Gamma \rrfloor, x : \llceil T \rrceil \qquad \llbracket W \rrbracket^{\mathsf{v}}_r \vdash \llfloor \Delta \rrfloor, r : \llceil U \rrceil}{\llbracket V \rrbracket^{\mathsf{v}}_x \parallel \llbracket W \rrbracket^{\mathsf{v}}_r \vdash \llfloor \Gamma \rrfloor, x : \llceil T \rrceil \parallel \llfloor \Delta \rrfloor, r : \llceil U \rrceil}}{r[x].(\llbracket V \rrbracket^{\mathsf{v}}_x \parallel \llbracket W \rrbracket^{\mathsf{v}}_r) \vdash \llfloor \Gamma \rrfloor, \llfloor \Delta \rrfloor, r : \llceil T \rrceil \otimes \llceil U \rrceil}$$

Part 2.

- **Case** ($V\ W$). We assume $\llbracket V \rrbracket^{\mathsf{v}}_{y'} : \llfloor \Gamma \rrfloor, y' : \llceil T \rrceil^{\bot} \parr (\mathbf{1} \otimes \llceil U \rrceil)$ and $\llbracket W \rrbracket^{\mathsf{v}}_{x'} : \llfloor \Delta \rrfloor, x' : \llceil T \rrceil$ and derive

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{r \leftrightarrow y \vdash y : \llceil T \rrceil \otimes (\bot \parr \llceil U \rrceil^{\bot}), r : \mathbf{1} \otimes \llceil U \rrceil}}{y\langle x \rangle.r \leftrightarrow y \vdash x : \llceil T \rrceil^{\bot}, y : \llceil T \rrceil \otimes (\bot \parr \llceil U \rrceil^{\bot}), r : \mathbf{1} \otimes \llceil U \rrceil} \qquad \mathcal{D}}{y\langle x \rangle.r \leftrightarrow y \parallel \llbracket V \rrbracket^{\mathsf{v}}_{y'} \parallel \llbracket W \rrbracket^{\mathsf{v}}_{x'} \vdash x : \llceil T \rrceil^{\bot}, y : \llceil T \rrceil \otimes (\bot \parr \llceil U \rrceil^{\bot}), r : 1 \otimes \llceil U \rrceil \parallel \llfloor \Gamma \rrfloor, y' : \llceil T \rrceil^{\bot} \parr (\mathbf{1} \otimes \llceil U \rrceil) \parallel \llfloor \Delta \rrfloor, x' : \llceil T \rrceil}}{(\nu y y')(y\langle x \rangle.r \leftrightarrow y \parallel \llbracket V \rrbracket^{\mathsf{v}}_{y'} \parallel \llbracket W \rrbracket^{\mathsf{v}}_{x'}) \vdash x : \llceil T \rrceil^{\bot}, r : \mathbf{1} \otimes \llceil U \rrceil \parallel \llfloor \Gamma \rrfloor, \llfloor \Delta \rrfloor, x' : \llceil T \rrceil}}{(\nu x x')(\nu y y')(y\langle x \rangle.r \leftrightarrow y \parallel \llbracket V \rrbracket^{\mathsf{v}}_{y'} \parallel \llbracket W \rrbracket^{\mathsf{v}}_{x'}) \vdash \llfloor \Gamma \rrfloor, \llfloor \Delta \rrfloor, r : \mathbf{1} \otimes \llceil U \rrceil}$$

  where $\mathcal{D}$ is the derivation

$$\dfrac{\llbracket V \rrbracket^{\mathsf{v}}_{y'} \vdash \llfloor \Gamma \rrfloor, y' : \llceil T \rrceil^{\bot} \parr (\mathbf{1} \otimes \llceil U \rrceil) \qquad \llbracket W \rrbracket^{\mathsf{v}}_{x'} \vdash \llfloor \Delta \rrfloor, x' : \llceil T \rrceil}{\llbracket V \rrbracket^{\mathsf{v}}_{y'} \parallel \llbracket W \rrbracket^{\mathsf{v}}_{x'} \vdash \llfloor \Gamma \rrfloor, y' : \llceil T \rrceil^{\bot} \parr (\mathbf{1} \otimes \llceil U \rrceil) \parallel \llfloor \Delta \rrfloor, x' : \llceil T \rrceil}$$

- **Case** ($\textbf{let}\ (x, y) = V\ \textbf{in}\ M$).
  We assume $\llbracket V \rrbracket^{\mathsf{v}}_{y'} : \llfloor \Gamma \rrfloor, y' : \llceil T \rrceil \otimes \llceil T' \rrceil$ and $\llbracket M \rrbracket^{\mathsf{m}}_r : \llfloor \Delta \rrfloor, x : \llceil T \rrceil^{\bot}, y : \llceil T' \rrceil^{\bot}, r : \mathbf{1} \otimes \llfloor U \rrfloor^{\bot}$ and derive

$$\dfrac{\dfrac{\dfrac{\llbracket M \rrbracket^{\mathsf{m}}_r \vdash \llfloor \Delta \rrfloor, x : \llceil T \rrceil^{\bot}, y : \llceil T' \rrceil^{\bot}, r : \mathbf{1} \otimes \llfloor U \rrfloor^{\bot}}{y(x).\llbracket M \rrbracket^{\mathsf{m}}_r \vdash \llfloor \Delta \rrfloor, y : \llceil T \rrceil^{\bot} \parr \llceil T' \rrceil^{\bot}, r : \mathbf{1} \otimes \llfloor U \rrfloor^{\bot}} \qquad \llbracket V \rrbracket^{\mathsf{v}}_{y'} \vdash \llfloor \Gamma \rrfloor, y' : \llceil T \rrceil \otimes \llceil T' \rrceil}{y(x).\llbracket M \rrbracket^{\mathsf{m}}_r \parallel \llbracket V \rrbracket^{\mathsf{v}}_{y'} \vdash \llfloor \Delta \rrfloor, y : \llceil T \rrceil^{\bot} \parr \llceil T' \rrceil^{\bot}, r : \mathbf{1} \otimes \llfloor U \rrfloor^{\bot} \parallel \llfloor \Gamma \rrfloor, y' : \llceil T \rrceil \otimes \llceil T' \rrceil}}{(\nu y y')(y(x).\llbracket M \rrbracket^{\mathsf{m}}_r \parallel \llbracket V \rrbracket^{\mathsf{v}}_{y'}) \vdash \llfloor \Gamma \rrfloor, \llfloor \Delta \rrfloor, r : \mathbf{1} \otimes \llfloor U \rrfloor^{\bot}}$$

- **Case** ($\mathbf{absurd}\ V$). We assume $\llbracket V \rrbracket^{\mathsf{v}}_{x'} : \lfloor\lfloor \Gamma \rfloor\rfloor, x' : \mathbf{0}$, and derive:

$$\dfrac{\dfrac{\dfrac{}{x \triangleright \{\} \vdash r : \mathbf{1} \otimes \lfloor\lfloor T \rfloor\rfloor^{\perp}, x : \top} \qquad \llbracket V \rrbracket^{\mathsf{v}}_{x'} \vdash \lfloor\lfloor \Gamma \rfloor\rfloor, x' : \mathbf{0}}{x \triangleright \{\} \parallel \llbracket V \rrbracket^{\mathsf{v}}_{x'} \vdash r : \mathbf{1} \otimes \lfloor\lfloor T \rfloor\rfloor^{\perp}, x : \top \parallel \lfloor\lfloor \Gamma \rfloor\rfloor, x' : \mathbf{0}}}{(\nu x x')(x \triangleright \{\} \parallel \llbracket V \rrbracket^{\mathsf{v}}_{x'}) \vdash \lfloor\lfloor \Gamma \rfloor\rfloor, r : \mathbf{1} \otimes \lfloor\lfloor T \rfloor\rfloor^{\perp}}$$

- **Case** ($\mathbf{let}\ x = M\ \mathbf{in}\ N$).
  We assume $\llbracket M \rrbracket^{\mathsf{m}}_{x'} : \lfloor\lfloor \Gamma \rfloor\rfloor, x' : \mathbf{1} \otimes \lceil\lceil T \rceil\rceil$ and $\llbracket N \rrbracket^{\mathsf{m}}_{r} : \lfloor\lfloor \Delta \rfloor\rfloor, x : \lceil\lceil T \rceil\rceil^{\perp}, r : \lceil\lceil U \rceil\rceil$ and derive

$$\dfrac{\dfrac{\dfrac{\llbracket N \rrbracket^{\mathsf{m}}_{r} \vdash \lfloor\lfloor \Delta \rfloor\rfloor, x : \lceil\lceil T \rceil\rceil^{\perp}, r : \lceil\lceil U \rceil\rceil}{x.\llbracket N \rrbracket^{\mathsf{m}}_{r} \vdash \lfloor\lfloor \Delta \rfloor\rfloor, x : \perp \parr \lceil\lceil T \rceil\rceil^{\perp}, r : \lceil\lceil U \rceil\rceil} \qquad \llbracket M \rrbracket^{\mathsf{m}}_{x'} \vdash \lfloor\lfloor \Gamma \rfloor\rfloor, x' : \mathbf{1} \otimes \lceil\lceil T \rceil\rceil}{x.\llbracket N \rrbracket^{\mathsf{m}}_{r} \parallel \llbracket M \rrbracket^{\mathsf{m}}_{x'} \vdash \lfloor\lfloor \Delta \rfloor\rfloor, x : \perp \parr \lceil\lceil T \rceil\rceil^{\perp}, r : \lceil\lceil U \rceil\rceil \parallel \lfloor\lfloor \Gamma \rfloor\rfloor, x' : \mathbf{1} \otimes \lceil\lceil T \rceil\rceil}}{(\nu x x')(x.\llbracket N \rrbracket^{\mathsf{m}}_{r} \parallel \llbracket M \rrbracket^{\mathsf{m}}_{x'}) \vdash \lfloor\lfloor \Gamma \rfloor\rfloor, \lfloor\lfloor \Delta \rfloor\rfloor, r : \lceil\lceil U \rceil\rceil}$$

- **Case** (V). We assume $\llbracket V \rrbracket^{\mathsf{v}}_{r} : \lfloor\lfloor \Gamma \rfloor\rfloor, r : \lceil\lceil T \rceil\rceil$ and derive

$$\dfrac{\llbracket V \rrbracket^{\mathsf{v}}_{r} \vdash \lfloor\lfloor \Gamma \rfloor\rfloor, r : \lceil\lceil T \rceil\rceil}{\bar{r}.\llbracket V \rrbracket^{\mathsf{v}}_{r} \vdash \lfloor\lfloor \Gamma \rfloor\rfloor, r : \mathbf{1} \otimes \lceil\lceil T \rceil\rceil}$$

Part 3. The cases are all by immediate induction. ☐

Lemma C.5. *Let F be an HGV∗ evaluation context and r a result endpoint. Then there exists a process context $\llbracket F \rrbracket^{\mathsf{f}}_{r}$ and a result endpoint $v = \mathsf{hr}(F, r)$ for the hole such that for all M we have that $\llbracket F[M] \rrbracket^{\mathsf{c}}_{r} = \llbracket F \rrbracket^{\mathsf{f}}_{r}[\llbracket M \rrbracket^{\mathsf{m}}_{v}]$.*

Proof. By induction on the structure of F. ☐

In the above lemma, if F is the empty context then $v = r$. Otherwise v is a variable bound by the process context $\llbracket F \rrbracket^{\mathsf{f}}_{r}$.

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

(1) *If $M \longrightarrow_{\mathsf{M}} M'$, then there exists a P such that $\llbracket M \rrbracket^{\mathsf{m}}_{r} \overset{\beta^{+}}{\Longrightarrow}_{\alpha} P$ and $P \approx_{\alpha} \llbracket M' \rrbracket^{\mathsf{m}}_{r}$; and*
(2) *if $\llbracket M \rrbracket^{\mathsf{m}}_{r} \xrightarrow{\beta} P$, then there exists an M' and a P' such that $M \longrightarrow_{\mathsf{M}} M'$ and $P \overset{\beta^{*}}{\Longrightarrow}_{\alpha} P'$ and $P' \approx_{\alpha} \llbracket M' \rrbracket^{\mathsf{m}}_{r}$.*

Proof.

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

Case (E-Lam).

$$\begin{tikzcd}
(\lambda x.M)\ V \arrow[rr, "\longrightarrow_{\mathsf{M}}"] \arrow[d, "\llbracket \cdot \rrbracket^{\mathsf{m}}_{r}"] & & M\{V/x\} \arrow[ddd, "\llbracket \cdot \rrbracket^{\mathsf{m}}_{r}"] \\
(\nu x x')(\nu y y')(y\langle x \rangle.r \leftrightarrow y \parallel y'(x).\llbracket M \rrbracket^{\mathsf{m}}_{y'} \parallel \llbracket V \rrbracket^{\mathsf{v}}_{x'}) \arrow[d, "\xrightarrow{\beta}\xrightarrow{\alpha}"] & & \\
(\nu x x')(\nu y y')(r \leftrightarrow y \parallel \llbracket M \rrbracket^{\mathsf{m}}_{y'} \parallel \llbracket V \rrbracket^{\mathsf{v}}_{x'}) \arrow[d, "\xrightarrow{\alpha}"] & & \\
(\nu x x')(\llbracket M \rrbracket^{\mathsf{m}}_{r} \parallel \llbracket V \rrbracket^{\mathsf{v}}_{x'}) \arrow[rr, dash, "\approx_{\alpha}\ \text{(by Lemma 5.10)}"] & & \llbracket M\{V/x\} \rrbracket^{\mathsf{m}}_{r}
\end{tikzcd}$$

Case (E-Unit).

$$\begin{tikzcd}
\textbf{let}\ () = ()\ \textbf{in}\ M \arrow[rr, "\longrightarrow_{\mathsf{M}}"] \arrow[d, "{[\![\cdot]\!]_r}"] & & M \arrow[dd, "{[\![\cdot]\!]_r}"] \\
(\nu x x')(x().[\![M]\!]_r^{\mathsf{m}} \parallel x'[].\mathbf{0}) \arrow[d, "\xrightarrow{\beta}"] & & \\
[\![M]\!]_r \parallel \mathbf{0} \arrow[rr, dash, "\sim"] & & [\![M]\!]_r^{\mathsf{m}}
\end{tikzcd}$$

Case (E-Pair).

$$\begin{tikzcd}
\textbf{let}\ (x, y) = (V, W)\ \textbf{in}\ M \arrow[rr, "\longrightarrow_{\mathsf{M}}"] \arrow[d, "{[\![\cdot]\!]_r}"] & & M\{V/x\}\{W/y\} \arrow[dd, "{[\![\cdot]\!]_r}"] \\
(\nu y y')(y(x).[\![M]\!]_r^{\mathsf{m}} \parallel y'[x'].([\![V]\!]_{x'}^{\mathsf{v}} \parallel [\![W]\!]_{y'}^{\mathsf{v}})) \arrow[d, "\xrightarrow{\beta}"] & & \\
(\nu y y')(\nu x x')([\![M]\!]_r \parallel [\![V]\!]_{x'}^{\mathsf{v}} \parallel [\![W]\!]_{y'}^{\mathsf{v}}) \arrow[rr, dash, "\approx_\alpha \text{ (by Lemma 5.10)}"] & & [\![M\{V/x\}\{W/y\}]\!]_r^{\mathsf{m}}
\end{tikzcd}$$

Case (E-Inl).

$$\begin{tikzcd}
\textbf{case inl}\ V\ \{\textbf{inl}\ x \mapsto M;\ \textbf{inr}\ y \mapsto N\} \arrow[rr, "\longrightarrow_{\mathsf{M}}"] \arrow[d, "{[\![\cdot]\!]_r}"] & & M\{V/x\} \arrow[dd, "{[\![\cdot]\!]_r}"] \\
(\nu x x')(x \triangleright \{\text{inl} : [\![M]\!]_r^{\mathsf{m}}; \text{inr} : [\![N\{x/y\}]\!]_r^{\mathsf{m}}\} \parallel x' \triangleleft \text{inl}.[\![V]\!]_{x'}^{\mathsf{v}}) \arrow[d, "\xrightarrow{\beta}"] & & \\
(\nu x x')([\![M]\!]_r \parallel [\![V]\!]_{x'}^{\mathsf{v}}) \arrow[rr, dash, "\approx_\alpha \text{ (by Lemma 5.10)}"] & & [\![M\{V/x\}]\!]_r^{\mathsf{m}}
\end{tikzcd}$$

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

Case (E-Let).

$$\begin{tikzcd}
\textbf{let}\ x = V\ \textbf{in}\ M \arrow[rr, "\longrightarrow_{\mathsf{M}}"] \arrow[d, "{[\![\cdot]\!]_r}"] & & M\{V/x\} \arrow[dd, "{[\![\cdot]\!]_r}"] \\
(\nu x x')(x.[\![M]\!]_r^{\mathsf{m}} \parallel \bar{x}'.[\![V]\!]_{x'}^{\mathsf{v}}) \arrow[d, "\xrightarrow{\beta}\xrightarrow{\beta}"] & & \\
(\nu x x')([\![M]\!]_r \parallel [\![V]\!]_{x'}^{\mathsf{v}}) \arrow[rr, dash, "\approx_\alpha \text{ (by Lemma 5.10)}"] & & [\![M\{V/x\}]\!]_r^{\mathsf{m}}
\end{tikzcd}$$

Case (E-Lift). The induction hypothesis yields the reasoning steps depicted by the first diagram, which we use, together with HGV's E-Lift and HCP's Str-Res and Str-Par2, to justify the second diagram:

$$\begin{tikzcd}
M \arrow[r, "\longrightarrow_{\mathsf{M}}"] \arrow[d, "\llbracket\cdot\rrbracket^{\mathsf{m}}_r"] & M' \arrow[d, "\llbracket\cdot\rrbracket^{\mathsf{m}}_r"] \\
\llbracket M\rrbracket^{\mathsf{m}}_r \arrow[r, "\stackrel{\beta^+}{\Longrightarrow}_\alpha\approx_\alpha"] & \llbracket M'\rrbracket^{\mathsf{m}}_r
\end{tikzcd}$$

$$\begin{tikzcd}
\textbf{let } x = E[M] \textbf{ in } N \arrow[r, "\longrightarrow_{\mathsf{M}}"] \arrow[d, "\llbracket\cdot\rrbracket_r"] & \textbf{let } x = E[M'] \textbf{ in } N \arrow[d, "\llbracket\cdot\rrbracket_r"] \\
(\nu x x')(x.\llbracket N\rrbracket^{\mathsf{m}}_r \parallel \llbracket M\rrbracket^{\mathsf{m}}_{x'}) \arrow[r, dash, "\stackrel{\beta^+}{\Longrightarrow}_\alpha\approx_\alpha"] & (\nu x x')(x.\llbracket N\rrbracket^{\mathsf{m}}_r \parallel \llbracket M'\rrbracket^{\mathsf{m}}_{x'})
\end{tikzcd}$$

(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 $(\nu x x')(\nu y y')(y\langle x\rangle.r{\leftrightarrow}y \parallel z{\leftrightarrow}y' \parallel \llbracket V\rrbracket^{\mathsf{v}}_{x'}) \stackrel{\beta}{\not\longrightarrow}$, which contradicts our premise. Therefore, $U = \lambda x.M$. The only possible β-transition is the one in the following diagram:

$$\begin{tikzcd}
(\lambda x.M)\ V \arrow[r, "\longrightarrow_{\mathsf{M}}"] \arrow[d, "\llbracket\cdot\rrbracket^{\mathsf{m}}_r"] & M\{V/x\} \arrow[ddd, "\llbracket\cdot\rrbracket^{\mathsf{m}}_r"] \\
(\nu x x')(\nu y y')(y\langle x\rangle.r{\leftrightarrow}y \parallel y'(x).\llbracket M\rrbracket^{\mathsf{m}}_{y'} \parallel \llbracket V\rrbracket^{\mathsf{v}}_{x'}) \arrow[d, "\stackrel{\beta}{\longrightarrow}\stackrel{\alpha}{\longrightarrow}"] & \\
(\nu x x')(\nu y y')(r{\leftrightarrow}y \parallel \llbracket M\rrbracket^{\mathsf{m}}_{y'} \parallel \llbracket V\rrbracket^{\mathsf{v}}_{x'}) \arrow[d, "\stackrel{\alpha}{\longrightarrow}"] & \\
(\nu x x')(\llbracket M\rrbracket^{\mathsf{m}}_r \parallel \llbracket V\rrbracket^{\mathsf{v}}_{x'}) \arrow[r, dash, "\approx_\alpha \text{ (by Lemma 5.10)}"] & \llbracket M\{V/x\}\rrbracket^{\mathsf{m}}_r
\end{tikzcd}$$

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

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

$$\begin{tikzcd}
\textbf{let } () = () \textbf{ in } M \arrow[r, "\longrightarrow_{\mathsf{M}}"] \arrow[d, "\llbracket\cdot\rrbracket_r"] & M \arrow[dd, "\llbracket\cdot\rrbracket_r"] \\
(\nu x x')(x().\llbracket M\rrbracket^{\mathsf{m}}_r \parallel x'[].\mathbf{0}) \arrow[d, "\stackrel{\beta}{\longrightarrow}"] & \\
\llbracket M\rrbracket_r \parallel \mathbf{0} \arrow[r, dash, "\sim"] & \llbracket M\rrbracket^{\mathsf{m}}_r
\end{tikzcd}$$

Hence, $M' = M$.

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

$$\begin{tikzcd}
\mathbf{let}\ (x,y) = (V,W)\ \mathbf{in}\ M \arrow[rr, "\longrightarrow_{\mathsf{M}}"] \arrow[d, "[\![\cdot]\!]_r"] & & M\{V/x\}\{W/y\} \arrow[dd, "[\![\cdot]\!]_r"] \\
(\nu y y')(y(x).[\![M]\!]_r^{\mathsf{m}} \parallel y'[x'].([\![V]\!]^{\mathsf{v}}_{x'} \parallel [\![W]\!]^{\mathsf{v}}_{y'})) \arrow[d, "\xrightarrow{\beta}"] & & \\
(\nu y y')(\nu x x')([\![M]\!]_r \parallel [\![V]\!]^{\mathsf{v}}_{x'} \parallel [\![W]\!]^{\mathsf{v}}_{y'}) \arrow[rr, dash, "\approx_\alpha \text{ (by Lemma 5.10)}"] & & [\![M\{V/x\}\{W/y\}]\!]^{\mathsf{m}}_r
\end{tikzcd}$$

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, $U = \mathbf{inl}\ V$. The only possible β-transition is the one in the following diagram:

$$\begin{tikzcd}
\mathbf{case}\ \mathbf{inl}\ V\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} \arrow[rr, "\longrightarrow_{\mathsf{M}}"] \arrow[d, "[\![\cdot]\!]_r"] & & M\{V/x\} \arrow[dd, "[\![\cdot]\!]_r"] \\
(\nu x x')(x \triangleright \{\mathrm{inl} : [\![M]\!]^{\mathsf{m}}_r; \mathrm{inr} : [\![N\{x/y\}]\!]^{\mathsf{m}}_r\} \parallel x' \triangleleft \mathrm{inl}.[\![V]\!]^{\mathsf{v}}_{x'}) \arrow[d, "\xrightarrow{\beta}"] & & \\
(\nu x x')([\![M]\!]_r \parallel [\![V]\!]^{\mathsf{v}}_{x'}) \arrow[rr, dash, "\approx_\alpha \text{ (by Lemma 5.10)}"] & & [\![M\{V/x\}]\!]^{\mathsf{m}}_r
\end{tikzcd}$$

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

Case ($\mathbf{let}\ x = M\ \mathbf{in}\ N$). 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:

$$\begin{tikzcd}
\mathbf{let}\ x = V\ \mathbf{in}\ M \arrow[rr, "\longrightarrow_{\mathsf{M}}"] \arrow[d, "[\![\cdot]\!]_r"] & & M\{V/x\} \arrow[dd, "[\![\cdot]\!]_r"] \\
(\nu x x')(x.[\![M]\!]^{\mathsf{m}}_r \parallel \bar{x}'.[\![V]\!]^{\mathsf{v}}_{x'}) \arrow[d, "\xrightarrow{\beta}\xrightarrow{\beta}"] & & \\
(\nu x x')([\![M]\!]_r \parallel [\![V]\!]^{\mathsf{v}}_{x'}) \arrow[rr, dash, "\approx_\alpha \text{ (by Lemma 5.10)}"] & & [\![M\{V/x\}]\!]^{\mathsf{m}}_r
\end{tikzcd}$$

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 Str-Res and Str-Par2.

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

Theorem 5.11 (Operational Correspondence)**.** *Suppose $\mathcal{C}$ is a well-typed configuration.*

(1) *(Preservation of reductions) If $\mathcal{C} \longrightarrow \mathcal{C}'$, then there exists a P such that $[\![\mathcal{C}]\!]^{\mathsf{c}}_r \stackrel{\beta^+}{\Longrightarrow}_\alpha P$ and $P \approx_\alpha [\![\mathcal{C}']\!]^{\mathsf{c}}_r$; and*
(2) *(Reflection of transitions)*
- *if $[\![\mathcal{C}]\!]^{\mathsf{c}}_r \xrightarrow{\alpha} P$, then $P \approx_\alpha [\![\mathcal{C}]\!]^{\mathsf{c}}_r$; and*

- *if* $\llbracket \mathcal{C} \rrbracket^{\mathsf{c}}_r \xrightarrow{\beta} P$*, then there exists a* $\mathcal{C}'$ *and a* P' *such that* $\mathcal{C} \longrightarrow \mathcal{C}'$ *and* $P \overset{\beta^*}{\Longrightarrow}_\alpha P'$ *and* $P' \approx_\alpha \llbracket \mathcal{C}' \rrbracket^{\mathsf{c}}_r$*. Furthermore,* $\mathcal{C}'$ *is unique up to structural congruence.*

Proof.

(1) By induction on the reduction $\mathcal{C} \longrightarrow \mathcal{C}'$. We implicitly make use of Lemma C.5 throughout the proof in order to recast the translation of a plugged evaluation context $\llbracket F[M] \rrbracket^{\mathsf{c}}_r$ into the plugging of the translated evaluation context with the translation of the plugged term $\llbracket F \rrbracket^{\mathsf{f}}_r[\llbracket M \rrbracket^{\mathsf{m}}_v]$ where $v = \mathsf{hr}(F, r)$.

Case (E-Reify-Fork).

$$
\begin{tikzcd}
F[\mathbf{fork}\ V] \arrow[rr, "\longrightarrow"] \arrow[d, "\llbracket\cdot\rrbracket^{\mathsf{c}}_r"] & & (\nu x x')(F[x] \parallel \circ\, V\ x') \arrow[dd, "\llbracket\cdot\rrbracket^{\mathsf{c}}_r"] \\
\llbracket F \rrbracket^{\mathsf{f}}_r[(\nu z z')(\nu y y')\begin{pmatrix} y\langle z\rangle.v\leftrightarrow y \parallel \\ (\nu x x')\begin{pmatrix} y'(w).x\langle w\rangle.\overline{y'}.y'\leftrightarrow x \parallel \\ x'(w).w\langle x'\rangle.w.w[].\mathbf{0} \end{pmatrix} \parallel \\ \llbracket V \rrbracket^{\mathsf{v}}_{z'} \end{pmatrix}] \arrow[d, "\xrightarrow{\beta}\xrightarrow{\beta}\xrightarrow{\alpha}"] & & \\
\llbracket F \rrbracket^{\mathsf{f}}_r[(\nu z z')\begin{pmatrix} (\nu x x')\begin{pmatrix} \bar{v}.v\leftrightarrow x \parallel \\ z\langle x'\rangle.z.z[].\mathbf{0} \end{pmatrix} \parallel \\ \llbracket V \rrbracket^{\mathsf{v}}_{z'} \end{pmatrix}] \arrow[r, dash, "\approx_\alpha"] & (\nu x x')\begin{pmatrix} \llbracket F \rrbracket^{\mathsf{f}}_r[\bar{v}.v\leftrightarrow x] \parallel \\ (\nu y y')\begin{pmatrix} (\nu w w')(\nu z z')\begin{pmatrix} z\langle w\rangle.y\leftrightarrow z \parallel \\ \llbracket V \rrbracket^{\mathsf{v}}_{z'} \parallel \\ w'\leftrightarrow x' \end{pmatrix} \parallel \\ y'.y'[].\mathbf{0} \end{pmatrix} \end{pmatrix} &
\end{tikzcd}
$$

The endpoint $v = \mathsf{hr}(F, r)$. The final two terms are bisimilar by Lemma C.4.

Case (E-Reify-Link).

$$
\begin{tikzcd}
\circ\, E[\mathbf{link}\ (x, y)] \arrow[rr, "\longrightarrow"] \arrow[d, "\llbracket\cdot\rrbracket^{\mathsf{c}}_r"] & & (\nu z z')(x \overset{z}{\leftrightarrow} y \parallel \circ\, E[z']) \arrow[dd, "\llbracket\cdot\rrbracket^{\mathsf{c}}_r"] \\
(\nu a a')(\llbracket E \rrbracket^{\mathsf{m}}_r[(\nu z z')(\nu w w')(w\langle z\rangle.v\leftrightarrow w \parallel w'(t).t(s).\overline{w'}.w'().s\leftrightarrow t \parallel z'\langle x\rangle.y\leftrightarrow z' \parallel \overline{a'}.a'[].\mathbf{0}))] \arrow[d, "\xrightarrow{\beta}\xrightarrow{\beta}\xrightarrow{\alpha}\xrightarrow{\alpha}"] & & \\
(\nu a a')(\llbracket E \rrbracket^{\mathsf{m}}_r[\bar{v}.v().x\leftrightarrow y] \parallel \overline{a'}.a'[].\mathbf{0}) \arrow[rr, dash, "\approx_\alpha"] & & (\nu z z')(\bar{z}.z().x\leftrightarrow y \parallel (\nu a a')(\llbracket E[v\leftrightarrow z'] \rrbracket^{\mathsf{m}}_a \parallel \overline{a'}.a'[].\mathbf{0}))
\end{tikzcd}
$$

The endpoint $v = \mathsf{hr}(F, r)$.

Case (E-Comm-Link).

$$
\begin{tikzcd}
(\nu z z')(\nu x x')(x \overset{z}{\leftrightarrow} y \parallel \circ\, z' \parallel \phi\ M) \arrow[r, "\longrightarrow"] \arrow[d, "\llbracket\cdot\rrbracket^{\mathsf{c}}_r"] & \phi\ (M\{y/x'\}) \arrow[dd, "\llbracket\cdot\rrbracket^{\mathsf{c}}_r"] \\
(\nu z z')(\nu x x')(\bar{z}.z().x\leftrightarrow y \parallel (\nu w w')(z'\leftrightarrow w \parallel w'.w'[].\mathbf{0}) \parallel \llbracket \phi\ M \rrbracket^{\mathsf{c}}_r) \arrow[d, "{\xrightarrow{\alpha}, \xrightarrow{\beta} \times 3}"] & \\
\llbracket \phi\ M \rrbracket^{\mathsf{c}}_r\{y/x'\} \arrow[r, "\approx_\alpha"] & \llbracket \phi\ M\{y/x'\} \rrbracket^{\mathsf{c}}_r
\end{tikzcd}
$$

Case (E-Comm-Send).

$$
\begin{tikzcd}
(\nu x x')(F[\textbf{send }(V, x)] \parallel F'[\textbf{recv } x']) \arrow[r, "\longrightarrow"] \arrow[d, "\llbracket\cdot\rrbracket^{\mathsf{c}}_r"] & (\nu x x')(F[x] \parallel F'[(V, x')]) \arrow[ddd, "\llbracket\cdot\rrbracket^{\mathsf{c}}_r"] \\
(\nu x x')\left(\begin{array}{l} \llbracket F\rrbracket^{\mathsf{f}}_r[(\nu y y')(\nu z z')(z\langle y\rangle.u{\leftrightarrow}z \parallel z'(t).t(s).t\langle s\rangle.\bar{z}'.z'{\leftrightarrow}t \parallel y'[w].(\llbracket V\rrbracket^{\mathsf{v}}_w \parallel x{\leftrightarrow}y'))] \parallel \\ \llbracket F'\rrbracket^{\mathsf{c}}_r[(\nu y y')(\nu z z')(z\langle y\rangle.v{\leftrightarrow}z \parallel z'(s).s(t).\bar{z}'.z'\langle t\rangle.z'{\leftrightarrow}s \parallel x'{\leftrightarrow}y')] \end{array}\right) \arrow[d, "\xrightarrow{\beta}\times 5, \xrightarrow{\alpha}\times 2"] & \\
(\nu x x')(\llbracket F\rrbracket^{\mathsf{f}}_r[(x\langle w\rangle.\bar{u}.x{\leftrightarrow}u \parallel \llbracket V\rrbracket^{\mathsf{v}}_w)] \parallel \llbracket F'\rrbracket^{\mathsf{c}}_r[x'(t).\bar{v}.v\langle t\rangle.v{\leftrightarrow}x']) \arrow[dd, "\beta"] & \\
& (\nu x x')(\llbracket F\rrbracket^{\mathsf{f}}_r[\bar{u}.x{\leftrightarrow}u] \parallel \llbracket F'\rrbracket^{\mathsf{c}}_r[\bar{v}.v[w].(\llbracket V\rrbracket^{\mathsf{v}}_w \parallel v{\leftrightarrow}x')]) \arrow[dl, dash, "\approx_\alpha"] \\
(\nu x x')(\llbracket F\rrbracket^{\mathsf{f}}_r[(\bar{u}.u{\leftrightarrow}x \parallel \llbracket V\rrbracket^{\mathsf{v}}_w)] \parallel \llbracket F'\rrbracket^{\mathsf{c}}_r[\bar{v}.v\langle w\rangle.v{\leftrightarrow}x']) &
\end{tikzcd}
$$

The endpoint $u = \mathsf{hr}(F, r)$ and the endpoint $v = \mathsf{hr}(F', r)$.

Case (E-Comm-Close).

$$
\begin{tikzcd}
(\nu x x)(\circ\, x \parallel F[\textbf{wait } x']) \arrow[r, "\longrightarrow"] \arrow[d, "\llbracket\cdot\rrbracket^{\mathsf{c}}_r"] & F[()] \arrow[dd, "\llbracket\cdot\rrbracket^{\mathsf{c}}_r"] \\
(\nu x x)\left(\begin{array}{l} (\nu y y')(\bar{y}.x{\leftrightarrow}y \parallel y'.y'[].\mathbf{0}) \parallel \\ \llbracket F\rrbracket^{\mathsf{f}}_r[(\nu z z')(\nu w w')(w\langle z\rangle.v{\leftrightarrow}w \parallel w'(s).s().\bar{w}'.w'[].\mathbf{0} \parallel x'{\leftrightarrow}z')] \end{array}\right) \arrow[d, "\xrightarrow{\beta}\times 3, \xrightarrow{\alpha}\times 3, \xrightarrow{\beta}"] & \\
\llbracket F\rrbracket^{\mathsf{f}}_r[\bar{v}.v[].\mathbf{0}] \arrow[r, "="] & \llbracket F\rrbracket^{\mathsf{f}}_r[\bar{v}.v[].\mathbf{0}]
\end{tikzcd}
$$

The endpoint $v = \mathsf{hr}(F, r)$.

Case (E-Res).

$$
\begin{tikzcd}
(\nu x y)\mathcal{C} \arrow[r, "\longrightarrow"] \arrow[d, "\llbracket\cdot\rrbracket^{\mathsf{c}}_r"] & (\nu x y)\mathcal{C}' \arrow[d, "\llbracket\cdot\rrbracket^{\mathsf{c}}_r"] \\
(\nu x y)\llbracket\mathcal{C}\rrbracket^{\mathsf{c}}_r \arrow[r, "\stackrel{\beta^+}{\Longrightarrow}_\alpha \approx_\alpha \text{(IH)}"] & (\nu x y)\llbracket\mathcal{C}'\rrbracket^{\mathsf{c}}_r
\end{tikzcd}
$$

Case (E-Par).

$$
\begin{tikzcd}
\mathcal{C} \parallel \mathcal{D} \arrow[r, "\longrightarrow"] \arrow[d, "\llbracket\cdot\rrbracket^{\mathsf{c}}_r"] & \mathcal{C}' \parallel \mathcal{D} \arrow[dd, "\llbracket\cdot\rrbracket^{\mathsf{c}}_r"] \\
\llbracket\mathcal{C}\rrbracket^{\mathsf{c}}_r \parallel \llbracket\mathcal{D}\rrbracket^{\mathsf{c}}_r \arrow[d, "\stackrel{\beta^+}{\Longrightarrow}_\alpha \approx_\alpha \text{(IH)}"] & \\
\llbracket\mathcal{C}'\rrbracket^{\mathsf{c}}_r \parallel \llbracket\mathcal{D}\rrbracket^{\mathsf{c}}_r \arrow[r, "="] & \llbracket\mathcal{C}' \parallel \mathcal{D}\rrbracket^{\mathsf{c}}_r
\end{tikzcd}
$$

Case (E-Equiv).

$$\begin{tikzcd}
\mathcal{C} \arrow[r, "\equiv"] \arrow[d, "\llbracket\cdot\rrbracket^{\mathsf{c}}_{r}"] & \mathcal{C}' \arrow[r, "\longrightarrow"] & \mathcal{D}' \arrow[r, "\equiv"] & \mathcal{E} \arrow[ddd, "\llbracket\cdot\rrbracket^{\mathsf{c}}_{r}"] \\
\llbracket\mathcal{C}\rrbracket^{\mathsf{c}}_{r} \arrow[d, "\approx_\alpha (\text{Lemma 5.6})"] & & & \\
\llbracket\mathcal{C}'\rrbracket^{\mathsf{c}}_{r} \arrow[d, "\stackrel{\beta^+}{\Longrightarrow}_\alpha \approx_\alpha (\text{IH})"] & & & \\
\llbracket\mathcal{D}'\rrbracket^{\mathsf{c}}_{r} \arrow[rrr, "\approx_\alpha (\text{Lemma 5.6})"] & & & \llbracket\mathcal{E}\rrbracket^{\mathsf{c}}_{r}
\end{tikzcd}$$

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

$$\begin{tikzcd}
\bullet M \arrow[r, "\longrightarrow"] \arrow[d, "\llbracket\cdot\rrbracket^{\mathsf{c}}_{r}"] & \bullet N \arrow[d, "\llbracket\cdot\rrbracket^{\mathsf{c}}_{r}"] \\
\llbracket M\rrbracket^{\mathsf{m}}_{r} \arrow[r, "\stackrel{\beta^+}{\Longrightarrow}_\alpha \approx_\alpha (\text{Lemma C.6})"] & \llbracket N\rrbracket^{\mathsf{m}}_{r}
\end{tikzcd}$$

(2) Reflection of α-transitions is trivial as α-transition is included in α-bisimulation. Reflection of β-transitions is by induction on C; as with Lemma C.6, the only well-typed β-transitions that can occur for each case are those specified in the simulation case. □