Separating Sessions Smoothly

Simon Fowler
University of Glasgow, UK

Wen Kokke
The University of Edinburgh, UK

Ornela Dardha
University of Glasgow, UK

Sam Lindley
The University of Edinburgh, UK

J. Garrett Morris
The University of Iowa, Iowa City, IA, USA

Abstract

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

2012 ACM Subject Classification Software and its engineering → Functional languages; Theory of computation → Linear logic

Keywords and phrases session types, hypersequents, linear lambda calculus

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2021.36

Related Version *Extended Version*: `https://arxiv.org/abs/2105.08996`

Funding Work 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.

Acknowledgements We thank the anonymous reviewers for their helpful comments and suggestions.

1 Introduction

Session types [18, 45, 19] are types used to verify communication protocols in concurrent and distributed systems: just as data types rule out dividing an integer by a string, session types rule out sending along an input channel. Session types originated in process calculi, but there is a gap between process calculi, which model the evolving state of concurrent systems, and the descriptions of these systems in typical programming languages. This paper addresses two foundations for session types: (1) a session-typed concurrent lambda calculus called GV [30], intended to be a modular and extensible basis for functional programming languages with session types; and, (2) a session-typed process calculus called CP [51], with a propositions-as-types correspondence to classical linear logic (CLL) [17].

Processes in CP correspond exactly to proofs in CLL and deadlock freedom follows from cut-elimination for CLL. However, while CP is strongly tied to CLL, at the same time it departs from π-calculus. Independent π-calculus features can only appear in combination in

CP: CP combines name restriction with parallel composition ($(\nu x)(P \parallel Q)$), corresponding to CLL's cut rule, and combines sending (of bound names only) with parallel composition ($x[y].(P \parallel Q)$), corresponding to CLL's tensor rule. This results in a proliferation of process constructors and prevents the use of standard techniques from concurrency theory, such as labelled-transition semantics and bisimulation, 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 [27] for a detailed discussion. Inspired by Carbone *et al.* [10] who used hypersequents [4] to give a logical grounding to choreographic programming languages [33], Hypersequent CP (HCP) [26, 27, 34] 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 [16], Lindley and Morris [30] describe GV's semantics by combining a reduction relation on single terms, following standard λ-calculus rules, and a reduction relation on concurrent configurations of terms, following standard π-calculus rules. They then give a semantic characterisation of deadlocked processes, an extrinsic [42] type system for configurations, and show that well-typed configurations are deadlock-free. There is, however, a large fly in this otherwise smooth ointment: process equivalence does not preserve typing. As a result, it is not enough for Lindley and Morris to show progress and preservation for well-typed configurations; instead, they must show progress and preservation for *all* configurations *equivalent to* well-typed configurations. This not only complicates the metatheory of GV, but the burden is inherited by any effort to build on GV's account of concurrency [14].

In this paper, we show that using hyper-environments in the typing of configurations enables a metatheory for GV that, compared to that of Lindley and Morris, is simpler, is more general, and as a result is easier to use and easier to extend. Hypersequent GV (HGV) repairs the treatment of process equivalence – equivalent configurations are equivalently typeable – and avoids the need for formal gimmickry connecting name restriction and parallel composition. HGV admits standard semantic techniques for concurrent programs: we use bisimulation to show that our translations both preserve *and reflect* reduction, whereas Lindley and Morris show only that their translations between GV and CP preserve reduction as well as resorting to weak explicit substitutions [28]. 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 [14].

Contributions. The paper contributes the following:

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

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

2 The Equivalence Embroglio

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

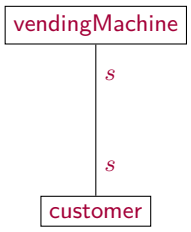
▶ **Example 2.1.** Consider the session type of a vending machine below, which sells candy bars and lollipops. If the vending machine is free, the customer can press ① to receive a candy bar or ② to receive a lollipop. If the vending machine is busy, the session ends.

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

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

```
let vendingMachine = λs.
  let s = select Free s in
  let s = offer s { ① ↦ send candyBar
                  ② ↦ send lollipop }
  close s
in let customer = λs.
  offer s { Free ↦ let s = select ① s in
                   let (cb, s) = recv s in
                   wait s; eat cb
          Busy ↦ wait s; hungry }
in let s = fork (λs.vendingMachine s)
in customer s
```

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



(b) Process structure of Figure 1a.

Figure 1 Example program with 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 [30] introduce a semantics for GV, which evaluates programs embedded in process configurations, consisting of embedded programs, flagged as main ($\bullet$) or child ($\circ$) threads, ν-binders to create new channels, and parallel compositions:

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

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

These restrictions are sufficient to guarantee deadlock freedom. Unfortunately, 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 [30]). *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 [14] or admit deadlocks [20, 46].

3 Hypersequent GV

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

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

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

The configuration language is reminiscent of π-calculus processes, but has some non-standard features. Name restriction uses double binders [49] in which one name is bound to each endpoint of the channel. Link threads [31] 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 [30], exploiting *hypersequents* [4] to recover modularity and extensibility. Inspired by HCP [34, 27, 26], configurations are typed under a *hyper-environment*, a collection of disjoint type environments. We let $\mathcal{G}, \mathcal{H}$ range over hyper-environments, writing $\varnothing$ for the empty hyper-environment, $\mathcal{G} \parallel \Gamma$ for disjoint extension of $\mathcal{G}$ with type environment Γ, and $\mathcal{G} \parallel \mathcal{H}$ for disjoint concatenation of $\mathcal{G}$ and $\mathcal{H}$.

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

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

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

Configuration types $R ::= \circ \mid \bullet\ T$

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

$$\bullet\ T \sqcap \circ = \bullet\ T \qquad \circ \sqcap \bullet\ T = \bullet\ T \qquad \circ \sqcap \circ = \circ$$

Figure 3 HGV, typing rules for configurations.

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

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

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

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

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

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

Figure 4 HGV, configuration reduction.

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

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

Figure 4 gives the configuration reduction rules. Thread contexts (F) extend evaluation contexts to threads, i. e., $F ::= \phi\ E$. The structural congruence rules are standard apart from SC-LinkComm, which ensures links are undirected, and SC-NewSwap, which swaps names in

double binders. The concurrent behaviour of HGV is given by a nondeterministic reduction relation ($\longrightarrow$) on configurations. The first two rules, E-Reify-Fork and E-Reify-Link, create child and link threads, respectively. The next three rules, E-Comm-Link, E-Comm-Send, and E-Comm-Close perform communication actions. The final four rules enable reduction under name restriction and parallel composition, rewriting by structural congruence, and term reduction in threads. Two rules handle links: E-Reify-Link creates a new *link thread* $x \overset{z}{\leftrightarrow} y$ which blocks on z of type $\mathbf{end}_?$, one endpoint of a fresh channel. The other endpoint, z' of type $\mathbf{end}_!$, is placed in the evaluation context of the parent thread. When z' terminates a child thread, E-Comm-Link performs forwarding by substitution.

Choice. Internal and external choice are encoded with sum types and session delegation [22, 13]. Prior encodings of choice in GV [30] are asynchronous. To encode synchronous choice we add a dummy synchronisation before exchanging the value of sum type, as follows:

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

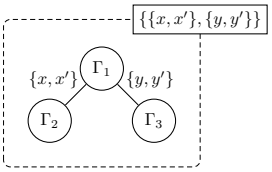
HGV enjoys type preservation, deadlock freedom, confluence, and strong normalisation (details in the extended version). Here we outline where the metatheory diverges from GV.

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

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

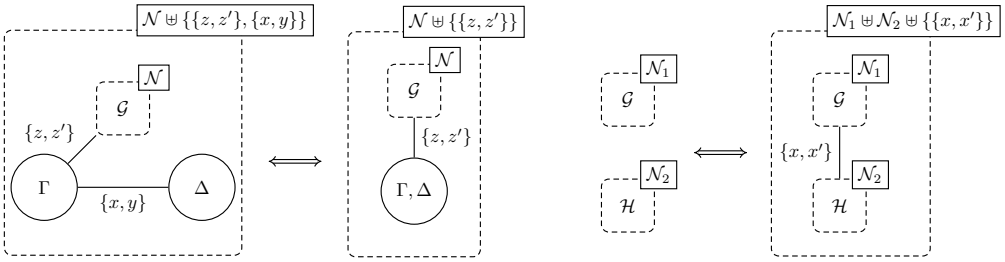
Abstract process structures. Unlike in GV, in HGV we cannot rely on the fact that exactly one channel is split over each parallel composition. Instead, we introduce the notion of an *abstract process structure* (APS). An APS is a graph defined over a hyper-environment $\mathcal{G}$ and a set of undirected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

▶ **Example 3.3.** Let $\mathcal{G} = \Gamma_1 \parallel \Gamma_2 \parallel \Gamma_3$, where $\Gamma_1 = x : S_1, y : S_2$, $\Gamma_2 = x' : \overline{S_1}, z : T$, and $\Gamma_3 = y' : \overline{S_2}$, and suppose $\mathcal{N} = \{\{x, x'\}, \{y, y'\}\}$. The APS for $\mathcal{G}$ and $\mathcal{N}$ is illustrated below.



A key feature of HGV is a subformula principle, which states that all hyper-environments arising in the derivation of an HGV program are tree-structured. We write $\mathsf{Tree}(\mathcal{G}, \mathcal{N})$ to denote that the APS for $\mathcal{G}$ with respect to $\mathcal{N}$ is tree-structured. An HGV program $\bullet\ M$ has

a single type environment, so is tree-structured; the same goes for child and link threads. Read bottom-up TC-New and TC-Par preserve tree structure (see the extended version for formal statements), which is illustrated by the following two pictures.



Tree canonical form. We now define a canonical form for configurations that captures the tree structure of an APS. Tree canonical form enables a succinct statement of *open progress* (Lemma 3.8) and a means for embedding HGV in GV (Lemma 4.5).

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

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

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

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

Progress and Deadlock Freedom.

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

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

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

1. *For each $\mathcal{A}_i$ $(1 \leq i \leq n)$, $\mathsf{blocked}(\mathcal{A}_i, z)$ 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 $z \in \{y_i \mid 1 \leq i \leq n\} \cup \mathrm{fv}(\Psi)$*

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

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 [8].

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

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

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

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

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

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

Figure 5 GV, typing rules for configurations.

4 Relation between HGV and GV

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

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

We require a pseudo-type $S^\sharp$, which types un-split channels. Un-split channels cannot appear in terms. 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 enforcing a tree process structure. The remaining typing rules are as in HGV.

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

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

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

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

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

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

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

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

We can write a GV typing derivation as follows:

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

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:

$$\dfrac{\dfrac{\dfrac{\Gamma_1, \Gamma_2, x : S \not\vdash_{\mathsf{GV}} \mathcal{C} \parallel \mathcal{D} : R_1 \sqcap R_2 \qquad \Gamma_3, x' : \overline{S}, \langle y, y'\rangle : S'^\sharp \not\vdash_{\mathsf{GV}} \mathcal{E} : R_3}{\Gamma_1, \Gamma_2, \Gamma_3, \langle x, x'\rangle : S^\sharp, \langle y, y'\rangle : S'^\sharp \not\vdash_{\mathsf{GV}} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} : R_1 \sqcap R_2 \sqcap R_3}}{\Gamma_1, \Gamma_2, \Gamma_3, \langle x, x'\rangle : S^\sharp \not\vdash_{\mathsf{GV}} (\nu yy')((\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}) : R_1 \sqcap R_2 \sqcap R_3}}{\Gamma_1, \Gamma_2, \Gamma_3 \not\vdash_{\mathsf{GV}} (\nu xx')(\nu yy')((\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}) : R_1 \sqcap R_2 \sqcap R_3}$$

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

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

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.

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

▶ 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.

▶ **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 correspondences.

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

$$\text{TP-Link}\ \frac{}{x \leftrightarrow^{A} y \vdash x : A, y : A^{\perp}} \qquad \text{TP-New}\ \frac{P \vdash \mathcal{G} \parallel \Gamma, x : A \parallel \Delta, y : A^{\perp}}{(\nu xy)P \vdash \mathcal{G} \parallel \Gamma, \Delta} \qquad \text{TP-Par}\ \frac{P \vdash \mathcal{G} \qquad Q \vdash \mathcal{H}}{P \parallel Q \vdash \mathcal{G} \parallel \mathcal{H}}$$

$$\text{TP-Halt}\ \frac{}{\mathbf{0} \vdash \varnothing} \qquad \text{TP-Close}\ \frac{P \vdash \varnothing}{x[].P \vdash x : \mathbf{1}}$$

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

$$\text{TP-Offer-Absurd}\ \frac{}{x \triangleright \{\} \vdash \Gamma, x : \top}$$

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

Duality $A^{\perp}$

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

Figure 6 HCP, duality and typing rules for processes.

Hypersequent CP. HCP [34, 27] 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 [44, 6], bound output [44], and double binders [49]. The syntax of processes is given by the typing rules (Figure 6), which are standard for HCP [34, 27]: $x \leftrightarrow y$ forwards messages between x and y; $(\nu xy)P$ creates a channel with endpoints x and y, and continues as P; $P \parallel Q$ composes P and Q in parallel; $\mathbf{0}$ is the terminated process; $x[y].P$ creates a new channel, outputs one endpoint over x, binds the other to y, and continues as P; $x(y).P$ receives a channel endpoint, binds it to y, and continues as P; $x[].P$ and $x().P$ close x and continue as P; $x \triangleleft \mathrm{inl}.P$ and $x \triangleleft \mathrm{inr}.P$ make a binary choice; $x \triangleright \{\mathrm{inl} : P; \mathrm{inr} : Q\}$ offers a binary choice; and $x \triangleright \{\}$ offers a nullary choice. As HCP is synchronous, the only difference between $x[y].P$ and $x(y).P$ is their typing (and similarly for $x[].P$ and $x().P$). We write *unbound* send as $x\langle y \rangle.P$ (short for $x[z].(y \leftrightarrow z \parallel P)$), and synchronisation as $\bar{x}.P$ (short for $x[z].(z[].\mathbf{0} \parallel P)$) and $x.P$ (short for $x(z).z().P$). Duality is standard and is involutive, i. e., $(A^{\perp})^{\perp} = A$.

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

Action rules

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

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

Communication Rules

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

$$\textsc{Bet-Send}\ \ \frac{P \xrightarrow{x[x']\|y(y')} P'}{(\nu xy)P \xrightarrow{\beta} (\nu xy)(\nu x'y')P'}$$

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

Structural Rules

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

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

Figure 7 HCP, label transition semantics.

We define the labelled transition system for HCP as a subsystem of that of Kokke et al. [26], omitting delayed actions. Labels ℓ represent the actions a process can take. Prefixes π are a convenient subset which can be written as prefixes to processes, i. e., $\pi.P$. The label τ represents internal actions. We distinguish two subtypes of internal actions: α represents only the evaluation of links as *renaming*, and β represents only *communication*.

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

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

The behavioural theory for HCP follows Kokke et al. [26], except that we distinguish two subrelations to bisimilarity, following the subtypes of internal actions.

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

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

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

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

Translating HGV to HCP. We factor the translation from HGV to HCP into two translations: (1) a translation into HGV$*$, a fine-grain call-by-value [29] variant of HGV, which makes control flow explicit; and (2) a translation from HGV$*$ to HCP.

HGV$*$. We define HGV$*$ as a refinement of HGV in which any non-trivial term must be named by a let binding before being used. While let is syntactic sugar in HGV, it is part of the core language in HGV$*$. Correspondingly, the reduction rule for let follows from the encoding in HGV, i. e.$\mathbf{let}\ x = V\ \mathbf{in}\ M \longrightarrow_{\mathsf{M}} M\{V/x\}$.

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	$::=$	$\Box \mid \mathbf{let}\ x = E\ \mathbf{in}\ M$

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\begin{array}{lll} \llparenthesis \mathbf{link} \rrparenthesis^{\mathsf{v}}_r = r(y).y(x).\bar{r}.r().x \leftrightarrow y & \llparenthesis \mathbf{send} \rrparenthesis^{\mathsf{v}}_r = r(y).y(x).y\langle x \rangle.\bar{r}.r \leftrightarrow y & \llparenthesis \mathbf{wait} \rrparenthesis^{\mathsf{v}}_r = r(x).x().\bar{r}.r[].\mathbf{0} \\ \llparenthesis \mathbf{fork} \rrparenthesis^{\mathsf{v}}_r = r(x).\bar{r}.x\langle r \rangle.x.x[].\mathbf{0} & \llparenthesis \mathbf{recv} \rrparenthesis^{\mathsf{v}}_r = r(x).x(y).\bar{r}.r\langle y \rangle.r \leftrightarrow x & \end{array}$$

Figure 8 Translation from HGV∗ to HCP.

Translating HCP to HGV. We cannot translate HCP processes to HGV terms directly: HGV's term language only supports **fork** (see the extended version 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.* [27, 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 [30, 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, 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.

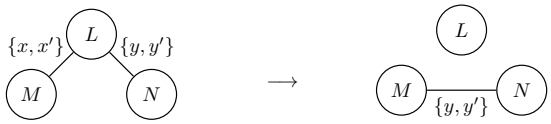
6 Extensions

In this section, we outline three extensions to HGV that exploit generalising the tree structure of processes to a forest structure. Full details are given in the extended version. 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 [30]. Other extensions to GV such as shared channels [30], polymorphism [32], and recursive session types [31] adapt to HGV almost unchanged.

From trees to forests. The TC-Mix structural rule allows two type environments Γ_1, Γ_2 to be split by a hyper-environment separator *without* a channel connecting them. Mix [17] may be interpreted as concurrency *without* communication [30, 3].

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

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

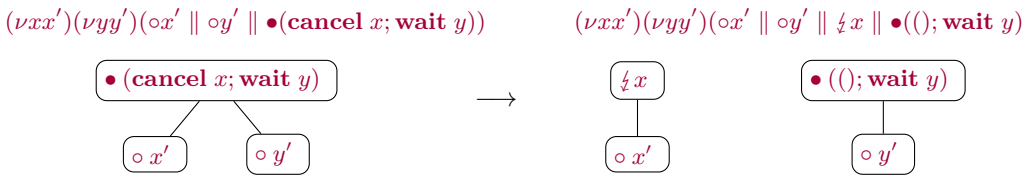


The result of link reduction has forest structure. Well-typed closed programs in both GV and HGV must *always* maintain tree structure. Different versions of GV do so in various unsatisfactory ways: one is pre-emptive blocking [30], which breaks confluence; another is two stage linking (Figure 4), which defers forwarding via a special link thread [31]. With TC-Mix, we can adjust the type schema for **link** to $(S \times \overline{S}) \multimap \mathbf{1}$ and use the following rule.

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

This formulation enables immediate substitution, maximimising concurrency.

Exceptions. In order to support exceptions in the presence of linear endpoints [14, 35] we must have a way of *cancelling* an endpoint ($\mathbf{cancel} : S \multimap \mathbf{1}$). Cancellation generates a special *zapper thread* ($\lightning\, x$) which severs a tree topology into a forest as in the following example.



7 Related work

Session Types and Functional Languages. HGV traces its origins to a line of work initiated by Gay and collaborators [15, 48, 50, 16]. This family of calculi builds session types directly into a lambda calculus. Toninho *et al.* [47] take an alternative approach, stratifying their

system into a session-typed process calculus and a separate functional calculus. There are many pragmatic embeddings of session type systems in existing functional programming languages [36, 41, 43, 21, 38, 24]. A detailed survey is given by Orchard & Yoshida [37].

Propositions as Sessions. When Girard introduced linear logic [17] he suggested a connection with concurrency. Abramsky [1] and Bellin and Scott [5] give embeddings of linear logic proofs in π-calculus, where cut reduction is simulated by π-calculus reduction. Both embeddings interpret tensor as parallel composition. The correspondence with π-calculus is not tight in that these systems allow independent prefixes to be reordered. Caires and Pfenning [7] give a propositions as types correspondence between dual intuitionistic linear logic and a session-typed π-calculus called πDILL. They interpret tensor as output. The correspondence with π-calculus is tight in that independent prefixes may not be reordered. With CP [51], Wadler adapts πDILL to classical linear logic. Aschieri and Genco [2] give an interpretation of classical multiplicative linear logic as concurrent functional programs. They interpret $\parr$ as parallel composition, and the connection to session types is less direct.

Priority-based Calculi. Systems such as πDILL, CP, and GV (and indeed HCP and HGV) ensure deadlock freedom by exploiting the type system to statically impose a tree structure on the communication topology – there can be at most one communication channel between any two processes. Another line of work explores a more liberal approach to deadlock freedom enabling some cyclic communication topologies, where deadlock freedom is guaranteed via *priorities*, which impose an order on actions. Priorites were introduced by Kobayashi and Padovani [23, 39] and adopted by Dardha and Gay [12] in Priority CP (PCP) and Kokke and Dardha in Priority GV (PGV) [25].

8 Conclusion and future work

HGV exploits hypersequents to resolve fundamental modularity issues with GV. As a consequence, we have obtained a tight operational correspondence between HGV and HCP. HGV is a modular and extensible core calculus for functional programming with *binary* session types. In future we intend to further exploit hypersequents in order to develop a modular and extensible core calculus for functional programming with *multiparty* session types. We would then hope to exhibit a similarly tight operational correspondence between this functional calculus and a multiparty variant of CP [9].

References

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

2 Federico Aschieri and Francesco A. Genco. Par means parallel: multiplicative linear logic proofs as concurrent functional programs. *Proc. ACM Program. Lang.*, 4(POPL):18:1–18:28, 2020.

3 Robert Atkey, Sam Lindley, and J. Garrett Morris. Conflation confers concurrency. In *A List of Successes That Can Change the World*, volume 9600 of *Lecture Notes in Computer Science*, pages 32–55. Springer, 2016.

4 Arnon Avron. Hypersequents, logical consequence and intermediate logics for concurrency. *Ann. Math. Artif. Intell.*, 4:225–248, 1991.

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

6 Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theoretical Computer Science*, 195(2):205–226, March 1998. doi:10.1016/s0304-3975(97)00220-x.

7 Luìs Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proc. of CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.

8 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.

9 Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *CONCUR*, volume 59 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

10 Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. *Distributed Comput.*, 31(1):51–67, 2018.

11 Olivier Danvy, Kevin Millikin, and Lasse R. Nielsen. On one-pass CPS transformations. *J. Funct. Program.*, 17(6):793–812, 2007.

12 Ornela Dardha and Simon J. Gay. A new linear logic for deadlock-free session-typed processes. In *Proc. of FoSSaCS*, volume 10803 of *LNCS*, pages 91–109. Springer, 2018.

13 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017.

14 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.

15 Simon J. Gay and Rajagopal Nagarajan. Intensional and extensional semantics of dataflow programs. *Formal Aspects of Computing*, 15(4):299–318, 2003.

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

17 Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

18 Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.

19 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.

20 Atsushi Igarashi, Peter Thiemann, Yuya Tsuda, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *J. Funct. Program.*, 29:e17, 2019.

21 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.

22 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.

23 Naoki Kobayashi. A new type system for deadlock-free processes. In *Proc. of CONCUR*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.

24 Wen Kokke and Ornela Dardha. Deadlock-free session types in linear Haskell. *CoRR*, abs/2103.14481, 2021. Accepted for publication at the Haskell Symposium 2021. arXiv:2103.14481.

25 Wen Kokke and Ornela Dardha. Prioritise the best variation. In *FORTE*, volume 12719 of *Lecture Notes in Computer Science*, pages 100–119. Springer, 2021.

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

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

28 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.

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

30 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.

31 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.

32 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.

33 Fabrizio Montesi. *Choreographic Programming*. PhD thesis, IT University of Copenhagen, 2013.

34 Fabrizio Montesi and Marco Peressotti. Classical transitions. *CoRR*, abs/1803.01049, 2018. arXiv:1803.01049.

35 Dimitris Mostrous and Vasco T. Vasconcelos. Affine sessions. *Log. Methods Comput. Sci.*, 14(4), 2018.

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

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

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

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

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

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

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

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

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

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

46 Peter Thiemann and Vasco T. Vasconcelos. Label-dependent session types. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–29, 2020.

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

48 Vasco Vasconcelos, Antonio Ravara, and Simon J. Gay. Session types for functional multithreading. In *CONCUR*, volume 3170 of *LNCS*, pages 497–511. Springer, 2004.

49 Vasco T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012.

50 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.

51 Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, 2014.