A Logical Interpretation of Asynchronous Multiparty Compatibility

Marco Carbone[1], Sonia Marin[2], and Carsten Schürmann[1]

[1] Computer Science Department, IT University of Copenhagen, Denmark
`{maca,carsten}@itu.dk`
[2] School of Computer Science, University of Birmingham, United Kingdom
`s.marin@bham.ac.uk`

Abstract Session types specify the protocols that communicating processes must follow in a concurrent system. When composing two or more processes, a session typing system must check whether such processes are *compatible*, i.e., that all sent messages are eventually received and no deadlock ever occurs. After the propositions-as-types paradigm, relating session types to linear logic, previous work has shown that *duality*, in the binary case, and more generally *coherence*, in the multiparty case, are sufficient syntactic conditions to guarantee compatibility for two or more processes, yet do not characterise all compatible set of processes. In this work, we generalise duality/coherence to a notion of *forwarder compatibility*. Forwarders are specified as a restricted family of proofs in linear logic, therefore defining a specific set of processes that can act as middleware by transfering messages without using them. As such, they can guide a network of processes to execute asynchronously. Our main result establishes forwarder compatibility as a sufficient and necessary condition to fully capture all well-typed multiparty compatible processes.

Keywords: Linear Logic · Session Types · Process Compatibility.

1 Introduction

Session types [16] are type annotations that ascribe protocols to processes in a concurrent system and determine how they communicate. *Binary session types* found a logical justification in *linear logic*, identified by Caires and Pfenning [4,3] and later by Wadler [27], which establishes the following correspondences: session types *as* linear logic propositions, processes *as* proofs, reductions in the operational semantics *as* cut reductions in linear logic, and duality *as* a notion of compatibility ensuring that two processes communication pattern match.

In binary session types, a sufficient condition for two protocols to be *compatible* in a synchronous execution is that their type annotations are dual: a send action of one party must match a corresponding receive action of the other party, and vice versa. Because of asynchronous interleavings, however, there are protocols that are compatible but not dual. The situation is even more complex for *multiparty session types* [17], which generalise binary session types for protocols with more than two participants. A central observation is that compatibility of sessions requires a property stronger than duality, ensuring that all messages sent by any participating party will eventually be collected by another. Deniélou and Yoshida [11] proposed the semantic notion of *multiparty compatibility.* The concept has then found many successful applications in the literature [11,21,12]. Yet, the question whether this notion "would be applicable to extend other theoretical foundations such as the correspondence with linear logic to multiparty communications" has not been answered since their original work.

As a first step in defining a logical correspondent to multiparty compatibility, Carbone et al. [8,5] extended Wadler's embedding of binary session types into classical linear logic (CLL) to the multiparty setting by generalising logical duality to the notion of *coherence* [17]. Coherence is also a sufficient compatibility condition: coherent processes are multiparty compatible, which ensures that their execution never leads to a communication error. Coherence is characterised proof-theoretically, and each coherence proof corresponds precisely to a multiparty protocol specification (*global type*), and in fact, coherence proofs correspond to a definable subset of the processes typable in linear logic, so-called *arbiters* [5]. In retrospect, the concept of coherence has sharpened our proof-theoretic understanding of how to characterise compatibility, but coherence, similarly to duality, cannot capture completely the notion of multiparty compatibility, i.e., there are compatible processes that are not coherent.

In this paper, we show that coherence (hence also duality) can be generalised to a notion of *forwarder compatibility.* Forwarders are processes that transfer messages between endpoints according to a protocol specification. Forwarders are more general than arbiters (every arbiter corresponds to a forwarder, not vice versa) but still can be used to guide the communication of multiple processes and guarantee they *communicate safely.* Our main result (Theorem 14) is that forwarders fully capture multiparty compatibility, which let us answer Deniélou and Yoshida's original question positively. In this work, we show that *i) any possible interleaving of a set of multiparty compatible processes can be encoded as a forwarder, and, conversely, ii) if a possible execution of a set of processes can be described by a forwarder, then such processes are indeed multiparty compatible.*

Forwarders are processes that dispatch messages: their behaviour can be seen as a specification, similarly to global types in multiparty session types. They capture the message flow by preventing messages from being *duplicated*, as superfluous messages would not be accounted for, and by preventing messages from being *lost*, otherwise a process might get stuck, awaiting a message. However, when data-dependencies permit, forwarders can choose to receive messages from different endpoints and forward such messages at a later point, or decide to buffer a certain number of messages. Eventually, they simply re-transmit messages after receiving them, *without* computing with them. Intuitively, this captures an interleaving of the communications between the given endpoints. Forwarders can be used to explain communication patterns as they occur in practice, including message routing, proxy services, and runtime monitors for message flows [19]. This paper shows that forwarders, as they capture multiparty compatibility, can supersede duality or coherence for composing processes. We achieve this logically: we generalise the linear logic cut rule with a new rule called MCutF which allows us to compose two or more processes (proofs) using a forwarder instead of duality [4] or coherence [5]. Our second main result is that MCutF can be eliminated by reductions that correspond to asynchronous process communications.

Contributions. The key contributions of this paper include: a definition of *multiparty compatibility* for classical linear logic (§ 3); a logical characterisation of *forwarders* that corresponds to multiparty compatibility (§ 4); and a composition mechanism (MCutF) for processes *with asynchronous communication* that uses forwarders and guarantees lack of communication errors (§ 5). Additionally, § 2 provides some background on types, processes, typing using linear logic, § 6 discusses related work, and concluding remarks and future work are in § 7.

All details, including proofs, can be found in our extended version [7].

2 CP and Classical Linear Logic

In this section, we give an introduction to Wadler's proposition-as-sessions approach [27], which comprises our variant of the CP language (Classical Processes) and its interpretation as sequent proofs in classical linear logic (CLL).

Types. Participants in a communication network are connected via endpoints acting as sockets where processes can write/read messages. Each endpoint is used according to its given session type describing how each endpoint must act.

Following the propositions-as-types approach, *types*, taken to be propositions (formulas) of CLL, denote the way an endpoint (a channel end) must be used at runtime. Their formal syntax is given by the following grammar:

$$\text{Types} \quad A ::= \quad a \mid a^{\perp} \mid \mathbf{1} \mid \perp \mid A \otimes A \mid A \mathbin{⅋} A \mid A \oplus A \mid A \mathbin{\&} A \mid !A \mid ?A \qquad (1)$$

Atoms a and negated atoms $a^{\perp}$ are basic dual types. Types $\mathbf{1}$ and $\perp$ denote an endpoint that must close with a last synchronisation. Type $A \otimes B$ is assigned to an endpoint that outputs a message of type A and then is used as B, and similarly, an endpoint of type $A \mathbin{⅋} B$, receives a message of type A and continues as B. In a branching choice, $A \oplus B$ is the type of an endpoint that may select to go left or right and continues as A or B, respectively, and $A \mathbin{\&} B$ is the type of an endpoint that offers two choices (left or right) and then, based on such choice, continues as A or B. Finally, $!A$ types an endpoint offering an unbounded number of copies of a service of type A, while $?A$ types an endpoint of a client invoking some replicated/unbounded service with behaviour A.

Duality. Operators can be grouped in pairs of duals that reflect the input-output duality. Consequently, standard duality $(\cdot)^{\perp}$ on types is inductively defined as:

$$(a^{\perp})^{\perp} = a \qquad \mathbf{1}^{\perp} = \perp \qquad (A \otimes B)^{\perp} = A^{\perp} \mathbin{⅋} B^{\perp} \quad (A \oplus B)^{\perp} = A^{\perp} \mathbin{\&} B^{\perp} \quad (!A)^{\perp} = ?A^{\perp}$$

In the remainder, for any binary operators $\oslash, \odot \in \{\otimes, ⅋, \oplus, \&\}$, we sometimes write $A \oslash B \odot C$ to mean $A \oslash (B \odot C)$.

Example 1 (Two-buyer protocol [17]). Two buyers intend to buy a book jointly from a seller. They are connecting through endpoints b_1, b_2 and s, respectively. The first buyer sends the title of the book to the seller, who, in turn, sends a quote to both buyers. Then, the first buyer decides how much she wishes to contribute and informs the second buyer, who either pays the rest or cancels by informing the seller. If the decision is to buy the book, the second buyer provides the seller with an address for shipping the book.

It is possible to type the first buyer's behaviour as $b_1 : \mathbf{name} \otimes \mathbf{cost}^{\perp} \parr \mathbf{cost} \otimes \mathbf{1}$ indicating that buyer b_1 first sends (expressed by $\otimes$) a value of type **name** (the book title), then receives (expressed by $\parr$) a value of type $\mathbf{cost}^{\perp}$ (the price of the book), then sends a value of type **cost** (the amount of money she wishes to contribute), and finally terminates. The behaviour of buyer b_2 and seller s can similarly be specified by session types, respectively $b_2 : \mathbf{cost}^{\perp} \parr \mathbf{cost}^{\perp} \parr ((\mathbf{addr} \otimes \mathbf{1}) \oplus \mathbf{1})$ and $s : \mathbf{name}^{\perp} \parr \mathbf{cost} \otimes \mathbf{cost} \otimes ((\mathbf{addr} \parr \perp) \,\&\, \perp)$.

Processes. As a language for *processes* we use a variant of the π-calculus [22] with specific communication primitives as standard for session calculi. Moreover, given that our theory is based on the proposition-as-sessions correspondence with CLL, we adopt a syntax akin to that of Wadler's CP [27]. For space reasons, we report the syntax of processes together with typing: each process can be found on the left-hand side of the turnstyle $\vdash$ in the conclusion of each rule in Figure 1. We briefly comment each process term. A link $x \leftrightarrow y$ is a binary forwarder, i.e., a process that forwards any communication between endpoints x and y. This yields a sort of equality relation on names: it says that endpoints x and y are equivalent, and communicating something over x is like communicating it over y. Note that we use endpoints instead of channels [26]. The terms $x().P$ and $x[]$ handle synchronisation (no message passing); $x().P$ can be seen as an empty input on x, while $x[]$ terminates the execution of the process. The term $x[y \triangleright P].Q$ denotes a process that creates a fresh name y (hence a new session), spawns a new process P, and then continues as Q. The intuition behind this communication operation is that P uses y as an interface for dealing with the continuation of the dual primitive (denoted by term $x(y).R$, for some R). Note that output messages are always fresh, as for the internal π-calculus [23], hence the output term $x[y \triangleright P].Q$ is a compact version of the π-calculus term $(\nu y)\, \overline{x}y.(P \mid Q)$. Branching computations are handled by $x.\mathsf{case}(P, Q)$, $x[\mathsf{inl}].P$ and $x[\mathsf{inr}].P$. The former denotes a process offering two options (external choice) from which some other process can make a selection with $x[\mathsf{inl}].P$ or $x[\mathsf{inr}].P$ (internal choice). Finally, $!x(y).P$ denotes a persistently available service that can be invoked by $?x[z].Q$ which will spawn a new session to be handled by a copy of process P.

Example 2 (Two-buyers, continued). For some contionuations P_i , Q_j, R_k, we provide possible implementations for the processes from the 2-buyer example, as

$$P_s \;=\; s(book).\; s[price \triangleright P_1].\; s[price \triangleright P_2].\; s.\mathsf{case}(s(addr).P_3, P_4),$$
$$P_{b_1} \;=\; b_1[book \triangleright Q_1].\; b_1(price).\; b_1[contr \triangleright Q_2].Q_3, \text{ and}$$
$$P_{b_2} \;=\; b_2(price).\; b_2(contr).\; b_2[\mathsf{inl}].b_2[addr \triangleright R_1].R_2$$

Note that the order in which the two buyers receive the price is not relevant.

$$\frac{}{x \leftrightarrow y \vdash x : a^{\perp}, y : a}\ \textsc{Ax} \qquad \frac{}{x[] \vdash x : \mathbf{1}}\ \mathbf{1} \qquad \frac{P \vdash \Delta}{x().P \vdash \Delta, x : \bot}\ \bot$$

$$\frac{P \vdash \Delta_1, y : A_1 \quad Q \vdash \Delta_2, x : A_2}{x[y \triangleright P].Q \vdash \Delta_1, \Delta_2, x : A_1 \otimes A_2}\ \otimes \qquad \frac{P \vdash \Delta, y : A_1, x : A_2}{x(y).P \vdash \Delta, x : A_1 \parr A_2}\ \parr$$

$$\frac{P \vdash \Delta, x : A_1}{x[\mathsf{inl}].P \vdash \Delta, x : A_1 \oplus A_2}\ \oplus_1 \qquad \frac{P \vdash \Delta, x : A_2}{x[\mathsf{inr}].P \vdash \Delta, x : A_1 \oplus A_2}\ \oplus_2$$

$$\frac{P \vdash \Delta, x : A_1 \quad Q \vdash \Delta, x : A_2}{x.\mathsf{case}(P, Q) \vdash \Delta, x : A_1 \,\&\, A_2}\ \& \qquad \frac{P \vdash\ ?\Delta, y : A}{!x(y).P \vdash\ ?\Delta, x : !A}\ !$$

$$\frac{P \vdash \Delta, y : A}{?x[y].P \vdash \Delta, x : ?A}\ ? \qquad \frac{P \vdash \Delta}{P \vdash \Delta, x : ?A}\ \textsc{w} \qquad \frac{P \vdash \Delta, y : ?A, z : ?A}{P\{x/y, x/z\} \vdash \Delta, x : ?A}\ \textsc{c}$$

Figure 1. Sequent Calculus for CP and Classical Linear Logic

CP-typing. Wadler [27] defined *Classical Processes* (CP) and showed that CLL proofs define a subset of well-behaved processes, satisfying deadlock freedom and session fidelity. Judgements are defined as $P \vdash \Delta$ with Δ a set of named types

$$\Delta ::= \quad \varnothing \mid x : A, \Delta$$

We interpret a judgement $P \vdash x_1 : A_1, \ldots, x_n : A_n$ as "P communicates on each endpoint x_i according to the protocol specified by A_i." System CP is given on Figure 1. It can be extended with a structural rule for defining composition of processes which corresponds to the Cut rule from CLL:

$$\frac{P \vdash \Delta_1, x : A \quad Q \vdash \Delta_2, y : A^{\perp}}{(\boldsymbol{\nu} xy)\,(P \mid Q) \vdash \Delta_1, \Delta_2}\ \textsc{Cut}$$

The process constructor corresponding to this rule is the *restriction* $(\boldsymbol{\nu} xy)$ which connects the two endpoints x and y. In CLL, this rule is admissible, i.e., cut-free derivations of the premises can be combined into a derivation of the conclusion with no occurrence of the Cut rule. This can then be extended into a constructive procedure, called *cut-elimination*, transforming a proof with cuts inductively into a cut-free proof. The strength of the proposition-as-type correspondence stems from the fact that it carries on to the proof level, since the cut-elimination steps correspond to computation in the form of reductions between processes [4,27]. In a multiparty setting, duality can be generalised and compatibility can be expressed as *coherence* [5].

However, not all compatible processes have dual types, as we can see in the following example.

Example 3 (Multiplicative criss-cross). Consider the two endpoints x and y willing to communicate with the following protocol – called a *criss-cross*: they both send a message to each other, and then the messages are received, according to the types $x : \mathbf{name} \otimes \mathbf{cost} \parr \mathbf{1}$ and $y : \mathbf{cost}^{\perp} \otimes \mathbf{name}^{\perp} \parr \bot$. Such protocol leads to no error (assuming asynchrony), still the two types above are not dual.

3 Multiparty Compatibility

Multiparty compatibility [11,21,12] allows for the composition of *multiple* processes while guaranteeing they will not get stuck or reach an error. It is a semantic notion that uses session types as an abstraction of process behaviours and simulates their execution. If no error occurs during any such simulation then the composition is considered compatible.

Extended types and queues. In order to define multiparty compatibility in the CLL setting, we extend the type syntax with annotations making explicit where messages should be forwarded from and to, similarly to local types $!\mathsf{p}.T$ and $?\mathsf{p}.T$ [10] expressing an output and an input to and from role p respectively. The meaning of each operator and the definition of duality remain as in CP.

$$\text{Local types} \qquad B, C ::= \quad a \mid a^{\perp} \mid \mathbf{1}^{\tilde{u}} \mid \perp^{u} \mid (A \otimes^{\tilde{u}} B) \mid (A \parr^{u} B)$$
$$\mid\ !^{\tilde{u}} B \mid ?^{u} B \mid (B \oplus^{u} C) \mid (B \&^{\tilde{u}} C)$$

Annotations are either single endpoints x or a set of endpoints $u_1, \ldots, u_n$, which we write as $\tilde{u}$ when its size is irrelevant. The left-hand side A of $\otimes$ and $\parr$ is a type as defined in (1) hence not annotated (but becomes dynamically so when needed). Units demonstrate some *gathering* behaviour which explains the need to annotate $\mathbf{1}$ with a non-empty list of distinct names. On the contrary, additives and exponential implement broadcasting: both & and ! are annotated with a non-empty list of distinct names.

After annotated types, to give a semantics to types, we introduce queues as

$$\Psi ::= \quad \epsilon \mid A \cdot \Psi \mid * \cdot \Psi \mid \mathcal{L} \cdot \Psi \mid \mathcal{R} \cdot \Psi \mid \mathcal{Q} \cdot \Psi$$

Intuitively, a queue (FIFO) is an ordered list of messages. A message can be a proposition A, a session termination $*$, a choice $\mathcal{L}$ or $\mathcal{R}$, or an exponential $\mathcal{Q}$. Every ordered pair of endpoints can be construed as haviing an associated queue. Hence, we formally define a *queue environment* σ as a mapping from ordered pairs of endpoints to queues: $\sigma : (x, y) \mapsto \Psi$. In the sequel, σ_ϵ denotes the queue environment with empty queues, while $\sigma[(x, y) \mapsto \Psi]$ denotes a new environment where the entry for (x, y) has been updated to Ψ. Finally, we define the type-context semantics for an annotated environment, i.e., an environment Δ where each formula is annotated (we abuse notation and overload the category Δ).

Definition 4 (Type-Context Semantics). *We define* $\xrightarrow{\alpha}$ *as the minimum relation of the form* $\Delta \bullet \sigma \xrightarrow{\alpha} \Delta' \bullet \sigma'$ *satisfying the following rules:*

$$\Delta, x : \perp^{y} \bullet \sigma[(x, y) \mapsto \Psi] \xrightarrow{x \perp y} \Delta \bullet \sigma[(x, y) \mapsto \Psi \cdot *]$$
$$x : \mathbf{1}^{\tilde{y}} \bullet \sigma_\epsilon[\{(y_i, x) \mapsto *\}_i] \xrightarrow{\tilde{y}\mathbf{1}x} \varnothing \bullet \sigma_\epsilon$$
$$x : a^{\perp}, y : a \bullet \sigma_\epsilon \xrightarrow{x \leftrightarrow y} \varnothing \bullet \sigma_\epsilon$$
$$\Delta, x : A \parr^{y} B \bullet \sigma[(x, y) \mapsto \Psi] \xrightarrow{x \parr y} \Delta, x : B \bullet \sigma[(x, y) \mapsto \Psi \cdot A]$$

$$\Delta, x : A \otimes^{\tilde{y}} B \bullet \sigma[\{(y_i, x) \mapsto A_i \cdot \Psi_i\}_i] \xrightarrow{\tilde{y} \otimes x[A, \{A_i\}_i]} \Delta, x : B \bullet \sigma[\{(y_i, x) \mapsto \Psi_i\}_i]$$
$$\Delta, x : B \,\&^{\tilde{y}}\, C \bullet \sigma[\{(x, y_i) \mapsto \Psi_i\}_i] \xrightarrow{x \&_{\mathcal{L}} \tilde{y}} \Delta, x : B \bullet \sigma[\{(x, y_i) \mapsto \Psi_i \cdot \mathcal{L}\}_i]$$
$$\Delta, x : B \,\&^{\tilde{y}}\, C \bullet \sigma[\{(x, y_i) \mapsto \Psi_i\}_i] \xrightarrow{x \&_{\mathcal{R}} \tilde{y}} \Delta, x : C \bullet \sigma[\{(x, y_i) \mapsto \Psi_i \cdot \mathcal{R}\}_i]$$
$$\Delta, x : B \oplus^{y} C \bullet \sigma[(y, x) \mapsto \mathcal{L} \cdot \Psi] \xrightarrow{y \oplus_{\mathcal{L}} x} \Delta, x : B \bullet \sigma[(y, x) \mapsto \Psi]$$
$$\Delta, x : B \oplus^{y} C \bullet \sigma[(y, x) \mapsto \mathcal{R} \cdot \Psi] \xrightarrow{y \oplus_{\mathcal{R}} x} \Delta, x : C \bullet \sigma[(y, x) \mapsto \Psi]$$
$$\{y_i : ?B_i\}_i, x : !^{\tilde{y}} C \bullet \sigma_\epsilon \xrightarrow{x ! \tilde{y}} \{y_i : ?B_i\}_i, x : C \bullet \sigma_\epsilon[\{(x, y_i) \mapsto \mathcal{Q}\}_i]$$
$$\Delta, x : ?^{\tilde{y}} B \bullet \sigma[(y, x) \mapsto \mathcal{Q} \cdot \Psi] \xrightarrow{y ? x} \Delta, x : B \bullet \sigma[(y, x) \mapsto \Psi]$$

where α ranges over labels denoting the type of action performed by the semantics, e.g., $\tilde{y}\mathbf{1}x$ signals an interaction from $\tilde{y}$ to x of type $\mathbf{1}$. Formally,

$$\begin{aligned} \alpha ::= \quad & x \perp y \mid \tilde{y}\mathbf{1}x \mid x \leftrightarrow y \mid x \parr y \mid \tilde{y} \otimes x[A, \{A_i\}_i] \\ & \mid x \&_{\mathcal{L}} \tilde{y} \mid x \&_{\mathcal{R}} \tilde{y} \mid y \oplus_{\mathcal{L}} x \mid y \oplus_{\mathcal{R}} x \mid x \,!\, \tilde{y} \mid y \,?\, x \end{aligned}$$

Intuitively, $\Delta \bullet \sigma \xrightarrow{\alpha} \Delta' \bullet \sigma'$ says that the environment Δ under the current queue environment σ performs α and becomes Δ' with updated queues σ'. The rules thus capture an asynchronous semantics for typing contexts.

Example 5. Assume we wish to compose three CP proofs through endpoints $\Delta = b_2 : \mathbf{cost}^\perp \parr B, b_1 : \mathbf{cost}^\perp \parr \mathbf{cost} \otimes \mathbf{1}, s : \mathbf{cost} \otimes C$. In order to obtain an execution of Δ, we first dualise and *choose* a way to annotate Δ as, e.g., $\Delta^\perp = b_2 : \mathbf{cost} \otimes^{b_1} B^\perp, b_1 : \mathbf{cost} \otimes^{s} \mathbf{cost}^\perp \parr^{b_2} \perp^{s}, s : \mathbf{cost}^\perp \parr^{b_2} C^\perp$. Then, from $\Delta^\perp \bullet \sigma_\epsilon$ we may obtain the execution

$$\xrightarrow{s \parr b_1} b_2 : \mathbf{cost} \otimes^{b_1} B^\perp, b_1 : \mathbf{cost} \otimes^{s} \mathbf{cost}^\perp \parr^{b_1} \perp^{s}, s : C \bullet \sigma_\epsilon[(s, b_1) \mapsto \mathbf{cost}^\perp]$$
$$\xrightarrow{s \otimes b_1[\mathbf{cost}, \mathbf{cost}^\perp]} b_2 : A \otimes^{b_2} B^\perp, b_1 : \mathbf{cost}^\perp \parr^{b_1} \perp^{s}, s : C \bullet \sigma_\epsilon$$
$$\xrightarrow{b_1 \parr b_2} b_2 : \mathbf{cost} \otimes^{b_1} B^\perp, b_1 : \perp^{s}, s : C \bullet \sigma_\epsilon[(b_1, b_2) \mapsto \mathbf{cost}^\perp]$$
$$\xrightarrow{b_1 \otimes b_2[\mathbf{cost}, \mathbf{cost}^\perp]} b_2 : B^\perp, b_1 : \perp^{s}, s : C \bullet \sigma_\epsilon$$

Note the general rule for the multiplicative connectors $\otimes$ and $\parr$. In their multiparty interpretation [5], they implement a gathering communication, where many $A_i \otimes B_i$ can communicate with a single $A \parr B$. As a consequence, the A_i's are enqueued to a single endpoint which will consume such messages. The effect of a gathering communication with such connectives is to spawn a new session with the environment $\{A_i\}_i$ shown in the label. Ideally, we could have enriched the semantics to work on different contexts running in parallel, where $\{A_i\}_i$ would be added to. However, since the semantics is used to define compatibility, we just observe the label. Units also have a similar gathering behaviour. On the other hand, additives and exponentials model broadcasting.

Using the relation on contexts above, we can define when a set of endpoints successfully progresses without reaching an error. This can be formalised by the concept of live path. In the sequel, let $\alpha_1, \ldots, \alpha_n$ ($\tilde{\alpha}$ for short) be a *path* for some annotated Δ whenever there exist $\Delta_1, \sigma_1, \ldots \Delta_n, \sigma_n$ such that $\Delta \bullet \sigma_\epsilon \xrightarrow{\alpha_1} \Delta_1 \bullet \sigma_1 \ldots \xrightarrow{\alpha_n} \Delta_n \bullet \sigma_n$. This path is *maximal* if there is no $\Delta_{n+1}, \sigma_{n+1}$ and α_{n+1} such that $\Delta_n \bullet \sigma_n \xrightarrow{\alpha_{n+1}} \Delta_{n+1} \bullet \sigma_{n+1}$.

Definition 6 (Live Path). *A path $\tilde{\alpha}$ for an environment $\Delta \bullet \sigma$ is* live *if*
$\Delta \bullet \sigma \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \varnothing \bullet \sigma_\epsilon$.

Intuitively, a maximal path is live whenever we can consume all send/receive operations specified in the type context and all queues are empty, i.e., an error is never reached. With this notion, we are ready to define multiparty compatibility.

Definition 7 (Multiparty Compatibility). *Environment $\Delta \bullet \sigma$ is* executable *if all maximal paths $\alpha_1, \ldots, \alpha_n$ for Δ are live and such that $\alpha_i = \tilde{y} \otimes x[A, \{A_i\}_i]$ implies $x : A^\perp, \{y_i : A_i^\perp\}_i$ is multiparty compatible for some annotation.*
A context $\Delta \neq \varnothing$ is multiparty compatible *if there exists an annotation such that $\Delta^\perp \bullet \sigma_\epsilon$ is executable.*

Multiparty compatibility states that *for a certain annotation* all maximal paths are live, i.e., no error ever occurs. This inductive definition is well-founded since propositions get smaller at each reduction.

Relationship to Previous Definitions. The original definition of compatibility given by Deniélou and Yoshida [11] was for communicating automata. Instead, Definition 4 is an adaptation of the *typing environment reduction* (Definition 4.3, [12]) with a little twist: in order not to overload notation, we are defining it on the dual of formulas. For example, similarly to [12], a process with endpoint x of type $A \otimes^y B$ stores something of type A in the queue from x to y. In our notation, we dualise the type of x to $A^\perp ⅋^y B^\perp$ but keep the same behaviour, i.e., storing something of type $A^\perp$ in the queue from x to y. Moreover, for the sake of simplicity, we are using a single queue environment σ as a function from pairs of endpoints to a FIFO, while [12] attaches labelled queues to each endpoint of the typing context: the two approaches are equivalent. Finally, our definition, being an adaptation to CLL, uses different language constructs. In particular, we do not combine value passing and branching, and $\otimes$ and ⅋ spawn new sessions (hence the well-founded recursive definition).

Properties of Multiparty Compatibility. As a consequence of multiparty compatibility, we can formalise the lack of errors with the following:

Proposition 8 (No Error). *Let Δ be multiparty compatible and $\alpha_1, \ldots, \alpha_n$ be a maximal path for an annotated $\Delta^\perp$ such that $\Delta^\perp \bullet \sigma_\epsilon \xrightarrow{\alpha_1} \Delta_1 \bullet \sigma_1 \ldots \xrightarrow{\alpha_n} \varnothing \bullet \sigma_\epsilon$. Then, for $i < n$,*

1. *(a) $\sigma_i(x, y) = * \cdot \Psi$ implies that $\alpha_n = x\tilde{z}\mathbf{1}y$;*
   (b) $\sigma_i(x, y) = A \cdot \Psi$ implies that there exists $k > i$ such that $\alpha_k = x\tilde{z} \otimes y[A, \{A_i\}_i]$;
   (c) $\sigma_i(x, y) = \mathcal{L} \cdot \Psi$ implies that there exists $k > i$ such that $\alpha_k = x \oplus_{\mathcal{L}} y$;
   (d) $\sigma_i(x, y) = \mathcal{R} \cdot \Psi$ implies that there exists $k > i$ such that $\alpha_k = x \oplus_{\mathcal{R}} y$;

   (e) $\sigma_i(x, y) = \mathcal{Q} \cdot \Psi$ *implies that there exists* $k > i$ *such that* $\alpha_k = x\,?\,y$;
2. *(a)* $\Delta_i = \Delta'_i, x : \mathbf{1}^{\tilde{y}}$, *then* $\alpha_n = \tilde{y}\mathbf{1}x$;
   (b) $\Delta_i = \Delta'_i, x : A \otimes^{\tilde{y}} B$, *then there exists* $k > i$ *such that* $\alpha_k = \tilde{y} \otimes x[\{A_i\}_i]$;
   (c) $\Delta_i = \Delta'_i, x : B \oplus^{y} C$, *then there exists* $k > i$ *such that* $\alpha_k = y \oplus_{\mathcal{L}} x$ *or* $\alpha_k = y \oplus_{\mathcal{R}} x$;
   (d) $\Delta_i = \Delta'_i, x : ?^{y} B$, *then there exists* $k > i$ *such that* $\alpha_k = y\,?\,x$.

Conditions in (1) state that every message that has been enqueued is eventually consumed, while conditions in (2) state that every input instruction is eventually executed. As CLL has no infinite behaviour, fairness conditions are not needed.

Are annotations important? A careful reader may be wondering why the definitions of type-context semantics and multiparty compatibility are not given for annotation-free contexts. Unfortunately, doing so would make multiparty compatibility too strong as it would allow for messages to be sent to different endpoints in *different paths*. As an example, $\Delta = b_2 : \mathbf{cost}^{\perp} \parr B, b_1 : \mathbf{cost}^{\perp} \parr \mathbf{cost} \otimes \mathbf{1}, s : \mathbf{cost} \otimes C$ can get stuck if s communicates with b_2 first, violating property (2b) in Proposition 8. Note that previous definitions of multiparty compatibility [21,12] indeed also use annotations.

4 Asynchronous Forwarders

Forwarders form a subclass of processes that are typable in classical linear logic. To identify them, we must add further information in the standard CP contexts.

Contexts. What we need is to be able to enforce the main features that characterise a forwarder, namely i) anything received must be forwarded, ii) anything that is going to be sent must be something that has been previously received, and iii) the order of messages between any two points must be preserved. In order to enforce these requirements, we add more information to the standard CP judgement. For example, let us consider the input process $x(y).P$. In CP, the typing environment for such process must be such that endpoint x has type $A \parr B$ such that P is typed in a context containing $y : A, x : B$. However, this does not make explicit that y is actually a message that has been received and, as such, should not be used by P for further communications but forwarded over some other channel. In order to remember this when we type the subprocess P, we insert $y : A$ into a queue that belongs to endpoint x where we put all the types of messages received over it. Namely, when typing P, the context will contain $[\![\Psi]\!][^{u}y : A]x : B$ meaning that x has type B and y type A in P, but moreover that $y : A$ has been received over x (it is in x's queue) and also that it is intended to be forwarded to endpoint u. In this setting, Ψ contains the types of messages that have been previously received over x. The forwarders behave asynchronously. They can input arbitrarily many messages, which are enqueued at the arrival point, without blocking the possibility of producing an output from the same endpoint. This behaviour is captured by the notion of queues of *boxed* messages, i.e. messages that are in-transit.

$$[\![\Psi]\!] ::= \varnothing \mid \quad [^{u}*][\![\Psi]\!] \mid \quad [^{u}y : A][\![\Psi]\!] \mid \quad [^{u}\mathcal{Q}][\![\Psi]\!] \mid \quad [^{u}\mathcal{L}][\![\Psi]\!] \mid \quad [^{u}\mathcal{R}][\![\Psi]\!]$$

$$\frac{}{x \leftrightarrow y \Vdash x : a^{\perp}, y : a}\ \mathrm{Ax} \qquad \frac{P \Vdash \mathit{\Gamma}, \llbracket \mathit{\Psi} \rrbracket [^{u}*] x : \cdot}{x().P \Vdash \mathit{\Gamma}, \llbracket \mathit{\Psi} \rrbracket x : \perp^{u}}\ \perp \qquad \frac{}{x[] \Vdash \{[^{x}*] u_i : \cdot\}_i, x : \mathbf{1}^{\tilde{u}}}\ \mathbf{1}$$

$$\frac{P \Vdash \mathit{\Gamma}, \llbracket \mathit{\Psi} \rrbracket [^{u} y : A] x : B}{x(y).P \Vdash \mathit{\Gamma}, \llbracket \mathit{\Psi} \rrbracket x : A \parr^{u} B}\ \parr \qquad \frac{P \Vdash \{y_i : A_i\}_i, y : A \quad Q \Vdash \mathit{\Gamma}, \{\llbracket \mathit{\Psi}_i \rrbracket u_i : A_i\}_i, \llbracket \mathit{\Psi} \rrbracket x : B}{x[y \triangleright P].Q \Vdash \mathit{\Gamma}, \{[^{x} y_i : A_i] \llbracket \mathit{\Psi}_i \rrbracket u_i : C_i\}_i, \llbracket \mathit{\Psi} \rrbracket x : A \otimes^{\tilde{u}} B}\ \otimes$$

$$\frac{P \Vdash \mathit{\Gamma}, \llbracket \mathit{\Psi} \rrbracket [^{\tilde{u}} \mathcal{L}] x : B \quad Q \Vdash \mathit{\Gamma}, \llbracket \mathit{\Psi} \rrbracket [^{\tilde{u}} \mathcal{R}] x : C}{x.\mathsf{case}(P, Q) \Vdash \mathit{\Gamma}, \llbracket \mathit{\Psi} \rrbracket x : B \,\&^{\tilde{u}}\, C}\ \&$$

$$\frac{P \Vdash \mathit{\Gamma}, \llbracket \mathit{\Psi}_z \rrbracket z : D, \llbracket \mathit{\Psi}_x \rrbracket x : B}{x[\mathsf{inl}].P \Vdash \mathit{\Gamma}, [^{x} \mathcal{L}] \llbracket \mathit{\Psi}_z \rrbracket z : D, \llbracket \mathit{\Psi}_x \rrbracket x : B \oplus^{z} C}\ \oplus_l \qquad \frac{P \Vdash \mathit{\Gamma}, \llbracket \mathit{\Psi}_z \rrbracket z : D, \llbracket \mathit{\Psi}_x \rrbracket x : C}{x[\mathsf{inr}].P \Vdash \mathit{\Gamma}, [^{x} \mathcal{R}] \llbracket \mathit{\Psi}_z \rrbracket z : D, \llbracket \mathit{\Psi}_x \rrbracket x : B \oplus^{z} C}\ \oplus_r$$

$$\frac{P \Vdash \{u_i : ?B_i\}_i, [^{\tilde{u}} \mathcal{Q}] y : C}{!x(y).P \Vdash \{u_i : ?B_i\}_i, x : !^{\tilde{u}} C}\ ! \qquad \frac{P \Vdash \mathit{\Gamma}, \llbracket \mathit{\Psi}_z \rrbracket z : C, \llbracket \mathit{\Psi}_x \rrbracket y : B}{?x[y].P \Vdash \mathit{\Gamma}, [^{x} \mathcal{Q}] \llbracket \mathit{\Psi}_z \rrbracket z : C, \llbracket \mathit{\Psi}_x \rrbracket x : ?^{z} B}\ ?$$

Figure 2. Proof System for Forwarders – in rules $\mathbf{1}$, $\otimes$, $\&$ and $!$, we ask that $\tilde{u} \neq \varnothing$

A queue element $[^{u} x : A]$ expresses that x of type A has been received and will need to later be forwarded to endpoint u. Similarly, $[^{u}*]$ indicates that a received request for closing a session must be forwarded to u. $[^{u}\mathcal{L}]$ (or $[^{u}\mathcal{R}]$) and $[^{u}\mathcal{Q}]$ indicate that a received branching request and server invocation, respectively, must be forwarded.

The order of messages needing to be forwarded to *independent* endpoints is irrelevant. Hence, we consider queue $\llbracket \mathit{\Psi}_1 \rrbracket [^{x} \dots][^{y} \dots] \llbracket \mathit{\Psi}_2 \rrbracket$ equivalent to queue $\llbracket \mathit{\Psi}_1 \rrbracket [^{y} \dots][^{x} \dots] \llbracket \mathit{\Psi}_2 \rrbracket$ whenever $x \neq y$. For a given endpoint x however the order of two messages $[^{x} \dots][^{x} \dots]$ is crucial and must be maintained throughout the forwarding. This follows the idea of having a queue for every ordered pair of endpoints in the type-context semantics in Definition 4. By attaching a queue to each endpoint we get a typing context

$$\mathit{\Gamma} ::= \ \varnothing \quad | \quad \mathit{\Gamma}, \llbracket \mathit{\Psi} \rrbracket x : B \quad | \quad \mathit{\Gamma}, \llbracket \mathit{\Psi} \rrbracket x : \cdot$$

The element $\llbracket \mathit{\Psi} \rrbracket x : B$ of a context $\mathit{\Gamma}$ indicates that the messages in $\llbracket \mathit{\Psi} \rrbracket$ have been received at endpoint x. The special case $\llbracket \mathit{\Psi} \rrbracket x : \cdot$ is denoting the situation when endpoint x no longer needs to be used for communication, but still has a non-empty queue of messages to forward.

When forwarding to many endpoints, we use $[^{\tilde{u}}\mathcal{X}]$ to denote $[^{u_1}\mathcal{X}] \dots [^{u_n}\mathcal{X}]$, with $\tilde{u} = u_1, \dots, u_n$. We also assume the implicit rewriting $[^{\varnothing}\mathcal{X}] \llbracket \mathit{\Psi} \rrbracket \equiv \llbracket \mathit{\Psi} \rrbracket$.

Judgements and rules. Judgement $P \Vdash \mathit{\Gamma}$ types the forwarder P connecting the endpoints in $\mathit{\Gamma}$. The rules for $\perp$, $\parr$, $\&$ and $!$ enforce asynchronous forwarding by adding elements to queues which are later dequeued by the corresponding rules for $\mathbf{1}$, $\otimes$, $\oplus$ or $?$. The rules are reported in Fig. 2.

Rule Ax is identical to the one of CP. Rules $\mathbf{1}$ and $\perp$ forward a request to close a session. Rule $\perp$ receives the request on endpoint x and enqueues it as $[^{u}*]$ if it needs to forward it to u. Note that in the premiss of $\perp$ the endpoint is terminated pending the remaining messages in the corresponding queue being dispatched. Eventually all endpoints but one will be terminated in the same manner. Rule $\mathbf{1}$ will then be applicable. Note that $x().P$ and $x[]$ behave as gathering, where several terminated endpoints connect to the last active one typed with a $\mathbf{1}$. Rules $\otimes$ and $\parr$ forward a message. Rule $\parr$ receives $y : A$ and enqueues it as $[^u y : A]$ to be forwarded to endpoint u. Dually, rule $\otimes$ sends the messages at the top of the queues of endpoints u_i's, meaning that several messages are sent at the same time. Messages will be picked from queues belonging to distinct endpoints, as a consequence, the left premiss of the $\otimes$-rule spawns a new forwarder (the gathered messages). In the case of additives and exponentials, the behaviour is *broadcasting*, i.e., an external choice $[^u\mathcal{L}]$ or $[^u\mathcal{R}]$, or a server opening $[^u\mathcal{Q}]$, resp., is received and can be used several (at least one) times to guide internal choices or server requests, resp., later on. Note how annotations put constraints on how proofs are constructed, e.g., annotating $x : A \parr B$ with u makes sure that the application of a $\parr$-rule for this formula will be followed by a $\otimes$-rule application on u later in the proof.

Example 9 (Two-buyers, continued). The forwarder process dispatching messages between the two buyers and the seller could be implemented as:

$$b'_1(book).\ s'[book \triangleright T_1].\ s'(price).\ s'(price).\ b'_1[price \triangleright T_2].\ b'_2[price \triangleright T_3].$$
$$b'_1(contr).\ b'_2[contr \triangleright T_4].b'_2.\mathsf{case}(s'[\mathsf{inl}].\ b'_2(addr).\ s[addr \triangleright T_5].\ T_6, T_7)$$

for some continuations T_i's. It captures the message flows between the different endpoints. Namely, it receive a name from b_1, forward it to s, and then proceed to receiving the price from s, forward it to b_1 and b_2, and so on.

The following example illustrates the need to support buffering, and reordering in order to capture the message flows between several processes.

Example 10 (Multiplicative criss-cross, continued). We can write a forwarder typable in the context $x : \mathbf{name}^\perp \parr \mathbf{cost}^\perp \otimes \bot, y : \mathbf{cost} \parr \mathbf{name} \otimes \mathbf{1}$ formed by the duals of the types in Example 3, i.e., a process that first receives on both x and y and then forwards the received messages over to y and x, respectively. $P := x(u).y(v).y[u' \triangleright u \leftrightarrow u'].x[v' \triangleright v' \leftrightarrow v].x().y[]$ is one of the forwarders that can prove the compatibility of the types involved in the criss-cross protocol, as illustrated by the derivation below.

$$\dfrac{\dfrac{\dfrac{\dfrac{F_1 := u \leftrightarrow u'}{\Vdash u : \mathbf{name}^\perp, u' : \mathbf{name}}\,\mathrm{Ax} \qquad \dfrac{\dfrac{F_2 := v' \leftrightarrow v}{\Vdash v' : \mathbf{cost}^\perp, v : \mathbf{cost}}\,\mathrm{Ax} \qquad \dfrac{\dfrac{F_3 := x().y[]}{y[] \Vdash [^y *]x : \cdot, y : \mathbf{1}}\,\mathbf{1}}{F_3 \Vdash x : \bot^y, y : \mathbf{1}^x}\,\bot}{x[v' \triangleright F_2].F_3 \Vdash x : \mathbf{cost}^\perp \otimes^y \bot^y, [^x v : \mathbf{cost}]y : \mathbf{1}^x}\,\otimes}{y[u' \triangleright F_1].x[v' \triangleright F_2].F_3 \Vdash [^y u : \mathbf{name}^\perp]x : \mathbf{cost}^\perp \otimes^y \bot^y, [^x v : \mathbf{cost}]y : \mathbf{name} \otimes^x \mathbf{1}^x}\,\otimes}{y(v).y[u' \triangleright F_1].x[v' \triangleright F_2].F_3 \Vdash [^y u : \mathbf{name}^\perp]x : \mathbf{cost}^\perp \otimes^y \bot^y, y : \mathbf{cost} \parr^x \mathbf{name} \otimes^x \mathbf{1}^x}\,\parr}{P \Vdash x : \mathbf{name}^\perp \parr^y \mathbf{cost}^\perp \otimes^y \bot^y, y : \mathbf{cost} \parr^x \mathbf{name} \otimes^x \mathbf{1}^x}\,\parr$$

Properties of Forwarders. We write $\llcorner B \lrcorner$ for the formula obtained from any B by removing all the annotations. We state that every forwarder is also a CP process, the embedding $\llcorner \cdot \lrcorner$ being extended to contexts and queues as:

$$\llcorner [\![\Psi]\!] x : B, \Gamma \lrcorner = \llcorner [\![\Psi]\!] \lrcorner, x : \llcorner B \lrcorner, \llcorner \Gamma \lrcorner \qquad \llcorner [\![\Psi]\!] x : \cdot, \Gamma \lrcorner = \llcorner [\![\Psi]\!] \lrcorner, \llcorner \Gamma \lrcorner$$

$$\llcorner [{}^u y : A][\![\Psi]\!] \lrcorner = y : A, \llcorner [\![\Psi]\!] \lrcorner \qquad \llcorner [{}^u \mathcal{X}][\![\Psi]\!] \lrcorner = \llcorner [{}^u *][\![\Psi]\!] \lrcorner = \llcorner [\![\Psi]\!] \lrcorner$$

$$\text{where } \mathcal{X} \in \{\mathcal{L}, \mathcal{R}, \mathcal{Q}\}$$

Proposition 11. *Any forwarder is typable in CP, i.e., if $P \Vdash \Gamma$, then $P \vdash \llcorner \Gamma \lrcorner$.*

Moreover, forwarders enjoy an invertibility property, i.e., in each rule if the conclusion is correct then so are the premisses. In CLL, the rules $\otimes$ or $\oplus$ are not invertible because of the choice involved either in splitting the context in the conclusion of $\otimes$ into the two premisses or the choice of either disjuncts for $\oplus$. In our case on the other hand, the annotations put extra syntactic constraints on what can be derived and hence are restricting these choices to a unique one and as a result the rules are invertible. This is formalised by the following.

Proposition 12. *All the forwarder rules are invertible, that is, for any rule if there exists a forwarder F such that $F \Vdash \Gamma$, the conclusion of the rule, there is a forwarder $F_i \Vdash \Gamma_i$, for each of its premisses, $i = 1$ or 2.*

Relation to Multiparty Compatibility. Forwarders relate to transitions in the type-context semantics introduced in the previous section. In order to formalise this, we first give a translation from type-contexts into forwarder contexts:

– $\mathsf{tr}(\varnothing \bullet \sigma_\epsilon) := \varnothing$;
– $\mathsf{tr}(\Delta, x : B \bullet \sigma\, [(y_i, x) \mapsto \Psi_i]_i) := [\![{}^{y_1} \Psi_1]\!] \ldots [\![{}^{y_n} \Psi_n]\!] x : B, \mathsf{tr}(\Delta \bullet \sigma)$
  for a type environment $\Delta = \{y_i : B_i\}_i$ and a queue environment σ mapping the endpoints y_i.

We use the notation $[\![{}^u \Psi]\!]$ to indicate that all brackets in $[\![\Psi]\!]$ are labelled by u.

Lemma 13. *Let $\Delta \bullet \sigma$ be a type-context and $\Gamma = \mathsf{tr}(\Delta \bullet \sigma)$.*

1. *if there exists α and $\Delta' \bullet \sigma'$ such that $\Delta \bullet \sigma \xrightarrow{\alpha} \Delta' \bullet \sigma'$ then there exists a rule in Fig. 2 such that Γ is an instance of its conclusion and $\Gamma' = \mathsf{tr}(\Delta' \bullet \sigma')$ is an instance of (one of) the premiss(es);*
2. *otherwise, either $\Delta = \varnothing$ and $\sigma = \sigma_\epsilon$ or there is no forwarder F such that $F \Vdash \Gamma$.*

We can conclude this section by stating that forwarders characterise multiparty compatibility for CP processes (processes that are well-typed in CLL).

Theorem 14. *Δ is multiparty compatible iff there exists a forwarder F such that $F \Vdash \Delta^\perp$ where each connective in $\Delta^\perp$ is annotated.*

Proof (Sketch). From left to right, we need to prove more generally that if $\Delta \bullet \sigma$ is executable, then there exists a forwarder F such that $F \Vdash \mathsf{tr}(\Delta \bullet \sigma)$, by induction on the size of Δ, defined as the sum of the formula sizes in Δ. From right to left can be proven by contrapositive, using Lemma 13 and Proposition 12. See [7] for the full proof.

5 Composing Processes with Asynchronous Forwarders

In this section, we show how to use forwarders to correctly compose CP processes.

Multiparty Process Composition. We start by focusing on the rule Cut, as seen in Section 2n which corresponds to parallel composition of processes. The implicit side condition that this rule uses is *duality*, i.e., we can compose two processes if endpoints x and y carry dual types.

Carbone et al. [5] introduced the concept of *coherence*, denoted by $\vDash$, which generalises duality to many endpoints, allowing for an extended cut-rule that composes many processes in parallel

$$\frac{\{R_i \vdash \Sigma_i, x_i : A_i\}_{i\leq n} \quad G \vDash \{x_i : A_i\}_{i\leq n}}{(\boldsymbol{\nu}\tilde{x} : G)\,(R_1 \;\mid\; \ldots \;\mid\; R_n) \vdash \{\Sigma_i\}_{i\leq n}}\ \text{MCut}$$

The judgement $G \vDash \{x_i : A_i\}_{i\leq n}$ intuitively says that the $x_i : A_i$'s are compatible and the execution of the R_i will proceed with no error. G is a process term and corresponds to a global type. A MCut elimination theorem analogous to the one of CP can be obtained.

Here, we replace coherence with an asynchronous forwarder Q, yielding rule

$$\frac{\{R_i \vdash \Sigma_i, x_i : A_i\}_{i\leq n} \quad Q \Vdash \left\{x_i : A_i^{\perp}\right\}_{i\leq n}}{(\boldsymbol{\nu}\tilde{x} : Q)\,(R_1 \;\mid\; \ldots \;\mid\; R_n) \vdash \{\Sigma_i\}_{i\leq n}}\ \text{MCutF}$$

Asynchronous forwarders are more general than coherence: every coherence proof can be transformed into an *arbiter* process [5], which is indeed a forwarder, while there are judgements that are not coherent but are provable in our forwarders (see Example 10). In the MCutF rule, the role of the forwarder is to be a middleware that decides whom to forward messages to. This means that when a process R_i sends a message, it must be stored by the forwarder, who will later forward it to the correct receiver. Our goal is to show that MCutF is admissible (and hence we can eliminate it from any correct proof). For this purpose, we need to extend the rule to also account for messages in transit, temporarily held by the forwarder. Making use of forwarders queues and some extra premisses, we define MCutQ as

$$\frac{\{P_j \vdash \Delta_j, y_j : A_j\}_{j\leq m} \quad \{R_i \vdash \Sigma_i, x_i : B_i\}_{i\leq n} \quad Q \Vdash \left\{[\![\Psi_i]\!]x_i : B_i^{\perp}\right\}_{i\leq n}, \{[\![\Psi_i]\!]x_i : \cdot\}_{n<i\leq p}}{(\boldsymbol{\nu}\tilde{x} : Q[\tilde{y} \triangleleft P_1, \ldots, P_m])\,(R_1 \;\mid\; \ldots \;\mid\; R_n) \vdash \{\Delta_j\}_{j\leq m}, \{\Sigma_i\}_{i\leq n}}$$

There are three kinds of process terms: P_j's, R_i's and Q. Processes R_i's are the ones communicating. Q is the forwarder who certifies compatibility, i.e., determine, at run time, who talks to whom. Finally, processes P_i's must be linked to messages in the forwarder queues. Such processes stem from the way $\otimes$ and $⅋$ work in linear logic as will become clearer when disucssing the reduction steps that lead to cut-admissibility. It imposes a side condition on the rule, namely that

$$\bigcup_{i\leq p} \Psi_i \setminus \{\mathcal{L}, \mathcal{R}, \mathcal{Q}, *\} = \left\{y_j : A_j^{\perp}\right\}_{j\leq m}$$

We need to introduce a new term syntax for this new structural rule: in the process $(\boldsymbol{\nu}\tilde{x} : Q[\tilde{y} \triangleleft P_1, \ldots, P_m])\,(R_1 \;\mid\; \ldots \;\mid\; R_n)$, the list $P_1, \ldots, P_m$ denotes those messages (processes) in transit that are going to form a new session after the communication has taken place. In the remainder we (slightly abusively) abbreviate both $\{P_1, \dots, P_m\}$ and $(R_1 \mid \dots \mid R_n)$ as $\tilde{P}$ and $\tilde{R}$ respectively.

Semantics and MCutF-admissibility. We now formally show that MCutF is admissible, yielding a semantics for our extended CP (with MCutF) in a proposition-as-types fashion. We illustrate the procedure on the multiplicative fragment (see [7] for all cases). In the sequel, we use the following abbreviations $\Gamma = \{[\![\Psi_i]\!]x_i : B_i^\perp\}_{i\leq n}, \{[\![\Psi_i]\!]x_i : \cdot\}_{n<i\leq p}$ and $\Gamma{-}k = \Gamma \setminus \{[\![\Psi_k]\!]x_k : B_k^\perp\}$. Also, we omit (indicated as "…") the premises of the MCutQ that do not play a role in the reduction at hand, and assume that they are always the same as above, that is, $\{P_j \vdash \Delta_j, y_j : A_j\}_{j\leq m}$ and $\{R_i \vdash \Sigma_i, x_i : B_i\}_{i\leq n}$.

Send Message ($\otimes$). This is the case when a process intends to send a message, which corresponds to a $\otimes$ rule. As a consequence, the forwarder has to be ready to receive and store the message (to forward it later):

$$\dfrac{\dfrac{P \vdash \Delta, y : A \quad R \vdash \Sigma, x : B}{x[y \triangleright P].R \vdash \Delta, \Sigma, x : A \otimes B}\otimes \qquad \dots \qquad \dfrac{Q \Vdash [\![\Psi]\!][^{x_k} y : A^\perp]x : B^\perp, \Gamma}{x(y).Q \Vdash [\![\Psi]\!]x : A^\perp \parr^{x_k} B^\perp, \Gamma}\parr}{(\boldsymbol{\nu} x\tilde{x} : x(y).Q[\tilde{y} \lhd \tilde{P}])\,(x[y \triangleright P].R \mid \tilde{R}) \vdash \Delta, \Sigma, \{\Delta_j\}_{j\leq m}, \{\Sigma_i\}_{i\leq n}}\text{MCutQ}$$

The process on the left is ready to send the message to the forwarder. By the annotation on the forwarder, it follows that the message will have to be forwarded to endpoint x_k, at a later stage. Observe that the nature of $\otimes$ is what introduces processes such as P to the rule: the idea is that when the forwarder will finalise the communication (by sending to a process R' owning endpoint x_k) process P will be composed with R'. For now, we obtain the reductum:

$$\dfrac{P \vdash \Delta, y : A \quad R \vdash \Sigma, x : B \qquad \dots \qquad Q \Vdash [\![\Psi]\!][^{x_k} y : A^\perp]x : B^\perp, \Gamma}{(\boldsymbol{\nu} x\tilde{x} : Q[y, \tilde{y} \lhd P, \tilde{P}])\,(R \mid \tilde{R}) \vdash \Delta, \Sigma, \{\Delta_j\}_{j\leq m}, \{\Sigma_i\}_{i\leq n}}\text{MCutQ}$$

Receive Message ($\parr$). At a later point, the forwarder will be able to complete the forwarding operation by connecting with a process ready to receive:

$$\dfrac{\dfrac{R \vdash \Sigma, y : A, x : B}{x(y).R \vdash \Sigma, x : A \parr B}\parr \quad \begin{array}{c} P \vdash \Delta, z : A^\perp \qquad\qquad \dots \\ \dfrac{S \Vdash z : A, y : A^\perp \quad Q \Vdash [\![\Psi_x]\!]x : B^\perp, \Gamma}{x[y \triangleright S].Q \Vdash [\![\Psi_x]\!]x : A^\perp \otimes^{x_k} B^\perp, [^{x} z : A][\![\Psi_k]\!]x_k : B_k^\perp, \Gamma - k}\otimes \end{array}}{(\boldsymbol{\nu} x\tilde{x} : x[y \triangleright S].Q[z, \tilde{y} \lhd P, \tilde{P}])\,(x(y).R \mid \tilde{R}) \vdash \Delta, \Sigma, \{\Delta_j\}_{j\leq m}, \{\Sigma_i\}_{i\leq n}}$$

This relies on process P with endpoint z of type $A^\perp$, endpoint x_k in the forwarder with a boxed endpoint z with type A, and process $x(y).R$ ready to receive.

After reduction, we obtain the following:

$$\dfrac{(\boldsymbol{\nu} yz : S)\,(R \mid P) \vdash \Sigma, \Delta, x : B \qquad \dots \qquad Q \Vdash [\![\Psi_x]\!]x : B^\perp, \Gamma}{(\boldsymbol{\nu} x\tilde{x} : Q[\tilde{y} \lhd \tilde{P}])\,((\boldsymbol{\nu} yz : S)\,(R \mid P) \mid \tilde{R}) \vdash \Delta, \Sigma, \{\Delta_j\}_{j\leq m}, \{\Sigma_i\}_{i\leq n}}\text{MCutQ}$$

Where the left premiss is obtained as follows:

$$\dfrac{R \vdash \Sigma, y : A, x : B \quad P \vdash \Delta, z : A^\perp \quad S \Vdash z : A, y : A^\perp}{(\boldsymbol{\nu} yz : S)\,(R \mid P) \vdash \Sigma, \Delta, x : B}\text{MCutQ}$$

meaning that now the message (namely process P) has finally been delivered and it can be directly linked to R with a new (but smaller) MCutQ.

These reductions (full set in [7]) let us prove the key lemma of this section.

Lemma 15 (MCutQ Admissibility). *If* $\{P_j \vdash \Delta_j, y_j : A_j\}_{j\leq m}$ *and* $\{R_i \vdash \Sigma_i, x_i : B_i\}_{i\leq n}$ *and* $Q \Vdash \{[\![\Psi_i]\!]x_i : B_i^\perp\}_{i\leq n}, \{[\![\Psi_i]\!]x_i : \cdot\}_{n<i\leq p}$ *there exists a process* $S \vdash \{\Delta_j\}_{j\leq m}, \{\Sigma_i\}_{i\leq n}$ *such that* $(\boldsymbol{\nu}\tilde{x} : Q[\tilde{y} \lhd \tilde{P}])\,\tilde{R} \Rightarrow^* S$.

Proof (Sketch). By lexicographic induction on (i) the sum of sizes of the B_i's and (ii) the sum of sizes of the R_i's. Some of the key cases have been detailed above; the others, as well as the base cases, can be found in the appendix of [7]. The commutative cases are straightforward and only need to consider the possible last rule applied to a premiss of the form $R_i \vdash \Sigma_i, x_i : B_i$.

We can finally conclude with the following theorem as a special case.

Theorem 16 (MCutF Admissibility). *If* $\{R_i \vdash \Sigma_i, x_i : A_i\}_{i\leq n}$ *and* $Q \Vdash \{x_i : A_i^\perp\}_{i\leq n}$ *then there exists a process* $S \vdash \{\Sigma_i\}_{i\leq n}$ *such that*
$(\boldsymbol{\nu}\tilde{x} : Q)\,(R_1 \mid \ldots \mid R_n) \Rightarrow^* S$.

6 Related Work

Our work takes [5] as a starting point. We set out to explore if coherence could be broken down into more elementary logical rules which led us to introduce forwarders, that turned out to provide a more general notion. An earlier version of this work [6] proposed synchronous forwarders: the restriction of forwarders with only buffers of size one. In that case, we show that we can always construct a coherence proof from a synchronous forwarder. However, synchronous forwarders fail to capture all the possible interleaving of an arbiter. The lack of global types in our work is strongly related to the work by Scalas and Yoshida [24], where compatibility and other properties are abstract away from the type system.

Caires and Perez [3] also study multiparty session types in the context of intuitionistic linear logic by translating global types to processes, called *mediums*. Their work does not start from a logical account of global types (their global types are just syntactic terms). But, as previous work [5], they do generate arbiters as linear logic proofs, which are special instances of forwarders. In a more recent work, van den Heuvel and Pérez [15] use *routers* in order to provide a decentralised analysis of multiparty protocols. Routers act as point-to-point forwarders but their types, called *relative types*, carry extra information on causality of events that are not local. In this work, we generalise both to characterise exactly which processes can justify the compatibility of a set of processes.

Another logical interpretation of multiparty compatibility is proposed by Horne [18]. which uses the additonal expressivity of BV, a generalisation of CLL with a non-commutative sequential operator but only a fragment expressible in the sequent calculus, unlike their previous work which relied on the full strength of deep inference [9]. It also allows one to consider compatible processes beyond duality but only for simply typed processes, which cannot spawn other processes. The main advantage of this approach is the fact that annotations are not needed.

Sangiorgi [23], probably the first to treat forwarders for the π-calculus, uses binary forwarders, i.e., processes that only forward between two channels, equivalent to our $x \leftrightarrow y$. In Caires and Pfenning [4], forwarders *à la Sangiorgi* were introduced as processes to be typed by the axiom rule in linear logic and we follow this tradition. Van den Heuvel and Perez [14] have recently developed a version of linear logic that encompasses both classical and intuitionistic logic, presenting a unified view on binary forwarders in both logics.

Barbanera and Dezani [2] study multiparty session types as *gateways* working as link forwarding communications between two multiparty sessions. Such mechanism reminds us of our forwarder composition: indeed, their related work mentions that gateways could be modelled by a "connection-cut".

Recent works [19,13] propose a variant of linear logic that models *identity providers*, monitors are similar to forwarders but restricted to binary sessions. They are asynchronous, i.e., allow for unbounded buffering of messages before forwarding. Our forwarders can be seen as a generalisation to multiparty.

7 Conclusions and Future Work

Forwarders are a logical characterisation of multiparty compatibility and they can safely replace coherence for composing any compatible processes. Below, we discuss some aspects of forwarders and identify possible extensions.

Improving Multiparty Compatibility? Multiparty compatibility concerns the error-free composition of processes by enqueeing/dequeueing messages into and from pair-wise distinct FIFO queues. We do not aim to improve multiparty compatibility, unlike, e.g., [12]. Rather, we assume a standard definition of it and give a logical characterisation, in the spirit of the approach started in [4]. The novelty is them to derive from the logical characterisation.

Are Forwarders Centralised? Following the approach taken for arbiters [5] and mediums [3], forwarders provide an orchestration of the message flows between the composed processes. To step to a fully decentralised setting would require to redefine rule MCut such that i) queues are no longer embedded in forwarders and ii) annotations in the forwarders are transferred to the composed processes. The correctness of these two steps follows from Theorem 14, since the type-context semantics in Definition 4 is indeed fully decentralised. Note that a similar decentralisation approach is also done for coherence in [5].

Process Language. Our process language is based on Wadler's CP [27], without polymorphic communications. We conjecture that forwarders can be extended to polymorphic types $\exists X.A$ and $\forall X.A$. We plan to consider a further extension to support recursion, inspired by Toninho et al. [25]. It would require an extended notion of compatibility dealing with infinite paths as done by Ghilezan et al. [12].

Variants of Linear Logic. Our theory is based on Classical LL for two main reasons. Coherence is indeed defined by Carbone et al. [5] in terms of CLL hence our results can be traced back to theirs. An early version of forwarders based on Intuitionistic Linear Logic (ILL) required many more rules, penalising the presentation. Nevertheless, our results should be adaptable to ILL. A different approach could be to include non-commutative operators which could encode our FIFO queues, e.g., non-commutative subexponentials by Kanovich et al. [20].

Beyond Linear Logic. Another interesting avenue would be to understand how the queueing mechanism of forwarders can be treated within a graphical proof system such as the one by Acclavio et al. [1]. Indeed, they observed that queues of length greater than 3 could not be expressed as linear logic formulas and thusdesigned a proof system that is based on general graphs.

Variants of Coherence. Our results show that forwarders generalise coherence proofs. Indeed, coherence would correspond to the notion of *synchronous forwarders*, the restriction of forwarders with only buffers of size one [6]. As a follow-up, we would like to investigate, whether other syntactic restrictions of forwarders also induce interesting generalised notions of coherence, and, as a consequence, generalisations of global types.

References

1. Acclavio, M., Horne, R., Mauw, S., Straßburger, L.: A graphical proof theory of logical time. In: Proc. of 7th Int. Conf. on Formal Structures for Computation and Deduction. LIPIcs, vol. 228 (2022)
2. Barbanera, F., Dezani-Ciancaglini, M.: Open multiparty sessions. In: Proc. of 12th Interaction and Concurrency Experience. EPTCS, vol. 304 (2019)
3. Caires, L., Pérez, J.A.: Multiparty session types within a canonical binary theory, and beyond. In: Proc. of Formal Techniques for Distributed Objects, Components, and Systems. LNCS, vol. 9688 (2016)
4. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Proc. of 21st Int. Conf. on Concurrency Theory. LNCS, vol. 6269 (2010)
5. Carbone, M., Lindley, S., Montesi, F., Schürmann, C., Wadler, P.: Coherence generalises duality: A logical explanation of multiparty session types. In: Proc. of 27th Int. Conf. on Concurrency Theory. LIPIcs, vol. 59 (2016)
6. Carbone, M., Marin, S., Schürmann, C.: Synchronous forwarders. CoRR **abs/2102.04731** (2021)
7. Carbone, M., Marin, S., Schürmann, C.: A logical interpretation of asynchronous multiparty compatibility. CoRR **abs/2305.16240** (2023)
8. Carbone, M., Montesi, F., Schürmann, C., Yoshida, N.: Multiparty session types as coherence proofs. In: Proc. of 26th Int. Conf. on Concurrency Theory. LIPIcs, vol. 42 (2015)
9. Ciobanu, G., Horne, R.: Behavioural analysis of sessions using the calculus of structures. In: Proc. of 10th Int. Andrei Ershov Memorial Conf. on Perspectives of System Informatics. LNCS, vol. 9609 (2015)
10. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global progress for dynamically interleaved multiparty sessions. MSCS **760** (2015)
11. Deniélou, P., Yoshida, N.: Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In: Proc. of 40th International Colloquium on Automata, Languages, and Programming. LNCS, vol. 7966 (2013)

12. Ghilezan, S., Pantovic, J., Prokic, I., Scalas, A., Yoshida, N.: Precise subtyping for asynchronous multiparty sessions. In: Proc. of 48th ACM Symp. on Principles of Program. Lang. vol. 5 (2021)
13. Gommerstadt, H., Jia, L., Pfenning, F.: Session-typed concurrent contracts. In: Proc. of 27th European Symp. on Programming. LNCS, vol. 10801 (2018)
14. van den Heuvel, B., Pérez, J.A.: Session type systems based on linear logic: Classical versus intuitionistic. In: Proc. of the 12th Int. Workshop on Prog. Lang. Approaches to Concurrency and Communication-cEntric Software. EPTCS, vol. 314 (2020)
15. van den Heuvel, B., Pérez, J.A.: A decentralized analysis of multiparty protocols. Science of Computer Programming **222** (2022)
16. Honda, K., Vasconcelos, V., Kubo, M.: Language primitives and type disciplines for structured communication-based programming. In: Proc. of 7th European Symp. on Programming. LNCS, vol. 1381 (1998)
17. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. Journal of the ACM **63**(1) (2016)
18. Horne, R.J.: Session subtyping and multiparty compatibility using circular sequents. In: Proc. of 31st Int. Conf. on Concurrency Theory. LIPIcs, vol. 171 (2020)
19. Jia, L., Gommerstadt, H., Pfenning, F.: Monitors and blame assignment for higher-order session types. In: Proc. of 43rd ACM Symp. on Principles of Programming Languages (2016)
20. Kanovich, M.I., Kuznetsov, S.L., Nigam, V., Scedrov, A.: A logical framework with commutative and non-commutative subexponentials. In: Proc. of 9th Int. Joint Conf. on Automated Reasoning. LNCS, vol. 10900 (2018)
21. Lange, J., Yoshida, N.: Verifying asynchronous interactions via communicating session automata. In: Proc. of 31st Int. Conf. on Computer Aided Verification. LNCS, vol. 11561 (2019)
22. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I and II. Information and Computation **100**(1) (1992)
23. Sangiorgi, D.: π-calculus, internal mobility, and agent-passing calculi. Theor. Comput. Sci. **167**(1-2) (1996)
24. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. In: Proc. of 46th ACM Symp. on Principles of Programming Languages. vol. 3 (2019)
25. Toninho, B., Caires, L., Pfenning, F.: Corecursion and non-divergence in session-typed processes. In: Proc. of 9th Int. Symp. on Trustworthy Global Computing. LNCS, vol. 8902 (2014)
26. Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. **217** (2012)
27. Wadler, P.: Propositions as sessions. J. of Functional Programming **24**(2–3) (2014)