Multiparty Session Types with a Bang!

Matthew Alan Le Brun, Simon Fowler, and Ornela Dardha

University of Glasgow
m.le-brun.1@research.gla.ac.uk
simon.fowler@glasgow.ac.uk
ornela.dardha@glasgow.ac.uk

Abstract. Replication is an alternative construct to recursion for describing infinite behaviours in the π-calculus. In this paper we explore the implications of including type-level replication in *Multiparty Session Types (MPST)*, a behavioural type theory for message-passing programs. We introduce MPST!, a session-typed multiparty process calculus with replication and first-class roles. We show that replication is *not* an equivalent alternative to recursion in MPST, and that using both replication *and* recursion in one type system in fact allows us to express both context-free protocols and protocols that support mutual exclusion and races. We demonstrate the expressiveness of MPST! on examples including binary tree serialisation, dining philosophers, and a model of an auction, and explore the implications of replication on the decidability of typechecking.

Keywords: Multiparty session types, Replication, Distributed protocols

1 Introduction

Our world is powered by a multitude of computer systems working together by *communicating*, *i.e.,* sending and receiving messages according to some *protocol*. It is therefore vital to *verify* the correctness of both communication protocols and their implementations, to ensure our programs behave according to their specifications, and to guarantee that specifications are indeed *safe*.

Session types [9,14,15,30] provide a lightweight method by which a developer can ensure safety of, and conformance to, communication protocols. Session types can be thought of as *types for protocols* which can be attached to a communication *channel* to specify *how* it should be used, and can be used to detect issues such as *communication mismatches* and *deadlocks* early in the development process. *Multiparty session types* (MPST) [7,16,24] generalise binary session types to allow reasoning about communication between two *or more* participants, and have been shown to be expressive enough to capture a range of practical protocols such as the OAuth 2 authentication protocol [26].

Example 1 (Client-Server-Worker). Using generalised MPST [26], we describe the types for three participants in a simple work-offloading system.

$$S_c := s \oplus \mathsf{req}(\mathtt{int}) \,.\, w \& \mathsf{ans}(\mathtt{str})$$
$$S_s := c \& \mathsf{req}(\mathtt{int}) \,.\, w \oplus \mathsf{fw}(\mathtt{int}) \qquad S_w := s \& \mathsf{fw}(\mathtt{int}) \,.\, c \oplus \mathsf{ans}(\mathtt{str}) \tag{1}$$

The above describes types for a *c*lient, *s*erver, and *w*orker respectively. The *c*lient, having type S_c, sends ($\oplus$) a request to the *s*erver with payload type int, then (.) waits to receive (&) an answer from the *w*orker with payload type str. The *s*erver, upon receiving the request from the *c*lient, forwards it to the *w*orker. Lastly the *w*orker, after receiving the forwarded request from the *s*erver, sends the answer to the *c*lient. A MPST system verifies that any written program code conforms to this specification (known as *session fidelity*), and that this protocol is *safe*—*i.e.,* that processes send and receive messages of compatible types.

Despite the potential of MPST for safe distributed programming, there remain limitations to the theory that impede their adoption for practical systems. For instance, generalising Example 1 to multiple workers in the style of a load-balancer is non-trivial and has inspired a series of work on the generalisation of *direction of choice* [27]. Further, generalising the number of *clients* is also non-trivial—typically, MPST theories assume a global view of *all* participants in a session. Lastly, the *objects* of actions in MPST (e.g., the recipient of a sent message) are *hard coded* because role names are *constants*.

Example 2 (Load Balancer for n clients). We introduce *replication* and *first-class roles*, combined with undirected choice in sends, to generalise Example 1 to support *two workers* and *any number* of clients.

$$\begin{aligned} S_s &:= !\alpha\&\mathsf{req}(\mathtt{int}) \;.\; \oplus \begin{cases} w_1\mathsf{fw}(\mathtt{int},\alpha) \;.\; \alpha\oplus\mathsf{wrk}(w_1) \\ w_2\mathsf{fw}(\mathtt{int},\alpha) \;.\; \alpha\oplus\mathsf{wrk}(w_2) \end{cases} \\ S_{w_i} &:= !s\&\mathsf{fw}(\mathtt{int},\gamma) \;.\; \gamma\oplus\mathsf{ans}(\mathtt{str}) \qquad \text{for } i \in \{1,2\} \end{aligned} \tag{2}$$

The first difference is the use of the *bang* (!) operator to denote a *replicated* action, *i.e.,* one which may occur *any number* of times. This makes the server agnostic to the *number* of requests. Second, a server now waits for requests—not from a specific client—but from *any* participant, binding the name of the sender to role variable α. This makes the server agnostic to the *source* of the requests. The server then makes a *choice* to forward the request to one of two workers, notably whilst passing the name of the client as one of the payloads. Finally, the server informs the client of the choice it made by sending the name of the worker in that branch. The worker type is also updated to be replicated, as it is dependent on the number of requests forwarded by the server. Notably, it receives the name of the client in the forward message, binding it to γ, and uses it to send the final answer. A client may now be defined as:

$$S_{c_j} := s\oplus\mathsf{req}(\mathtt{int}) \;.\; s\&\mathsf{wrk}(\omega) \;.\; \omega\&\mathsf{ans}(\mathtt{str}) \qquad \text{for any } j \in \mathbb{N} \tag{3}$$

As a result of the *replicated types* and *first-class role names* on the server-side, we may instantiate *any number of clients* and have them make *any number of requests*—all without changing the server-side protocol. Conversely, updating the number of workers has *no impact* on the types of clients. Thus, this extension promotes *modular* design of components in multiparty systems.

In fact, as we will see in Section 3, the addition of replication—especially when it is used in tandem with recursion—has several surprising consequences, in particular allowing us to describe *context-free protocols* as well as protocols that deal with *races* and *mutual exclusion*.

$$
\begin{array}{rll}
c ::= s[q] \mid x & & (\textit{session w/ role, channel variable}) \\
\rho ::= q \mid \alpha & & (\textit{role name value, role name variable}) \\
a ::= c \mid \alpha \quad V ::= c \mid \rho & & (\textit{names, values}) \\
b ::= x \mid \alpha \quad d ::= s[q] \mid q & & (\textit{binders, concrete values}) \\
P, Q ::= P \,|\, Q \mid (\nu s)\, P \mid \mathbf{0} & & (\textit{composition, restriction, termination}) \\
\mid \sum_{i \in I} c_i[\rho_i] \oplus \mathsf{m}_i\langle \widetilde{V_i} \rangle \,.\, P_i & & (\textit{choice of sends}) \\
\mid [!]\, c[\rho]\, \&_{i \in I}\, \mathsf{m}_i(\widetilde{b_i}) \,.\, P_i & & (\textit{[replicated] branching receive}) \\
\mid \mathsf{def}\ D\ \mathsf{in}\ Q \mid \mathrm{X}\langle \widetilde{c} \rangle & & (\textit{process definition, process call}) \\
D ::= \mathrm{X}(\widetilde{x}) = P & & (\textit{process declaration})
\end{array}
$$

Fig. 1. Syntax of MPST!

Contributions. The overarching contribution of this paper is the first integration of replication and first-class roles into a generalised MPST calculus, and an exploration of the impacts of these extensions on expressiveness and decidability. Our specific contributions are as follows:

1. We present MPST!, the first multiparty session-typed language with *replication* and *first-class roles* (Section 2), and prove its *metatheory* in the form of *subject reduction* and *session fidelity* properties (Section 2.4).
2. We show several expressiveness results through a series of representative examples (Section 3.1): in particular, replication lifts the expressive power of types and thus we give the first account of *context-free* MPST. We show that combining both replication and recursion allows us to model races and mutual exclusion; we demonstrate nontrivial examples including *binary tree serialisation*, the *dining philosophers problem*, and an *auction service*.
3. We demonstrate the impacts of replication on the decidability of typechecking (Section 3.2). We show that the decidability of typechecking is contingent on the decidability of a given safety property, and demonstrate conditions guaranteeing a property to be decidable. Finally we show two syntactic approximations to allow us to verify that a property is decidable.

Section 4 gives an account of related work and Section 5 concludes.

2 Multiparty Session Types with a Bang!

In this section we introduce MPST!, a conservative extension of existing multiparty session calculi [7,24,26] with support for *replication* and *first-class roles*.

2.1 Language

Figure 1 shows the syntax of MPST!.

Names, values, and binders. A *session name*, ranged over by $s, s', \ldots$, represents a collection of interconnected participants. A *role* is a participant in a multiparty communication protocol, and each *communication endpoint* $s[\boldsymbol{q}]$ is obtained by indexing a session name with a role. In contrast to existing MPST calculi, MPST! supports *first-class* roles, meaning that a role may be communicated as part of a message. To this end, a role $\boldsymbol{\rho}$ may either be a concrete role $\boldsymbol{p}$ (e.g., $\boldsymbol{s}$ or $\boldsymbol{w}$ in our load balancing example) or a *role variable* $\boldsymbol{\alpha}$. A name a is either an endpoint, a variable, or a role variable, whereas a value V is either a channel or a role. Binders b are used when receiving a message and can either be a variable binder or a role variable binder. Concrete values d are used when sending a message (at runtime) and are either an endpoint or a role name value.

Processes. Processes are ranged over by $P, Q, R, \ldots$: process $P \mid Q$ denotes P and Q running in parallel; *session restriction* $(\nu s)P$ binds session name s in process P; and $\mathbf{0}$ is the inactive process.

As in the π-calculus, but unlike other MPST calculi, MPST! supports *output-guarded choice* $\sum_{i\in I} c_i[\boldsymbol{\rho_i}] \oplus \mathsf{m}_i\langle\widetilde{V_i}\rangle \,.\, P_i$, allowing a nondeterministic send along any c_i to role $\boldsymbol{\rho_i}$ with label m_i and payload $\widetilde{V_i}$, with the process continuing as P_i. *Branching receive* $c[\boldsymbol{\rho}]\&_{i\in I}\ \mathsf{m}_i(\widetilde{b_i}) \,.\, P_i$ denotes a process waiting on channel c for one of a set of messages from role $\boldsymbol{\rho}$ with label m_i, binding the received data to $\widetilde{b_i}$ before continuing according to P_i. It is key to note that the object of a communication action is indicated via $\boldsymbol{\rho}$ (for both sending and receiving), which can either be a concrete value, or a role variable. The second key difference is the optional use of the *bang* ! with a branching receive, marking it as *replicated—i.e.,* it may be used 0 or more times, modelling *infinitely available servers.*

Definition 1 (Reduction context). *A* reduction context $\mathbb{C}$ *is given as:* $\mathbb{C} ::= \mathbb{C} \mid P \ \big|\ (\nu s)\,\mathbb{C} \ \big|\ \textsf{\textit{def}}\ D\ \textsf{\textit{in}}\ \mathbb{C} \ \big|\ [\,]$.

A *reduction context* allows us to evaluate processes under parallel composition and name restrictions. With this, reduction rules on processes are given in Figure 2. The rules make use of a standard *structural congruence* $\equiv$ (Appendix A) that allows us to treat parallel composition as commutative and associative, as well as including the usual π-calculus scope extrusion rule.

Rule [R-C] shows synchronous communication between two processes in session s. The first process, playing role $\boldsymbol{p}$, offers role $\boldsymbol{q}$ a choice of message labels and associated process continuations. The second process, playing role $\boldsymbol{q}$, sends a message with label m_k and transmits payloads $\widetilde{d}$. The first process reduces to the selected continuation with the transmitted payloads substituted for the binders in the selected branch, and the second process reduces to the continuation Q.

Rules [R-!C$_1$] and [R-!C$_2$] describe communication with a *replicated* process R. Rule [R-!C$_1$] is similar to R-C but the replicated process remains unchanged and the continuation Q_k is evaluated in parallel. Rule [R-!C$_2$] handles the case where the replicated process does not need to receive from a specific role, but instead allows communication with an *arbitrary* role: the rule binds the sending role to $\boldsymbol{\alpha}$ in the replicated continuation. We refer to this as a *universal receive.*

Process reduction $\boxed{P_1 \to P_2}$

R-C
$$s[\mathbf{p}][\mathbf{q}] \,\&_{i\in I}\, \mathsf{m}_i(\widetilde{b_i}) \,.\, P_i \mid s[\mathbf{q}][\mathbf{p}] \oplus \mathsf{m}_k\langle\widetilde{d}\rangle \,.\, Q \;\to\; P_k\{\widetilde{d}/\widetilde{b_k}\} \mid Q \quad \text{if } k \in I$$

R-!C$_1$
$$\frac{R = !s[\mathbf{p}][\mathbf{q}] \,\&_{i\in I}\, \mathsf{m}_i(\widetilde{b_i}) \,.\, Q_i}{s[\mathbf{q}][\mathbf{p}] \oplus \mathsf{m}_k\langle\widetilde{d}\rangle \,.\, P \mid R \;\to\; P \mid R \mid Q_k\{\widetilde{d}/\widetilde{b_k}\}} \quad \text{if } k \in I$$

R-!C$_2$
$$\frac{R = !s[\mathbf{p}][\boldsymbol{\alpha}] \,\&_{i\in I}\, \mathsf{m}_i(\widetilde{b_i}) \,.\, Q_i}{s[\mathbf{q}][\mathbf{p}] \oplus \mathsf{m}_k\langle\widetilde{d}\rangle \,.\, P \mid R \;\to\; P \mid R \mid Q_k\{\widetilde{d}/\widetilde{b_k}\}\{\mathbf{q}/\boldsymbol{\alpha}\}} \quad \text{if } k \in I$$

R-+
$$\textstyle\sum_{i\in I} s_i[\mathbf{q_i}][\mathbf{p_i}] \oplus \mathsf{m}_i\langle\widetilde{d_i}\rangle \,.\, P_i \;\to\; s_j[\mathbf{q_j}][\mathbf{p_j}] \oplus \mathsf{m}_j\langle\widetilde{d_j}\rangle \,.\, P_j \quad \text{for } j \in I$$

R-X
$$\mathsf{def}\ \mathrm{X}(\widetilde{x}) = P\ \mathsf{in}\ (\mathrm{X}\langle\widetilde{s'[\mathbf{r}]}\rangle \mid Q) \;\to\; \mathsf{def}\ \mathrm{X}(\widetilde{x}) = P\ \mathsf{in}\ (P\{\widetilde{s'[\mathbf{r}]}/\widetilde{x}\} \mid Q)$$

R-≡
$$\frac{P \equiv P' \qquad P \to Q \qquad Q \equiv Q'}{P' \to Q'}$$

R-$\mathbb{C}$
$$\frac{P \to P'}{\mathbb{C}[P] \to \mathbb{C}[P']}$$

Fig. 2. Operational semantics for the extended multiparty session π-calculus.

The [R-+] rule evaluates a branching output by nondeterministically evaluating to one of the sending branches; rule [R-X] handles a recursive call. Finally, rules [R-≡] and [R-$\mathbb{C}$] are administrative, allowing reduction modulo structural congruence and under contexts respectively.

Example 3 (Load Balancer: Process Reduction). We recall the load balancer example from Section 1, but this time, we present processes for each role (Figure 3) to demonstrate how our operational semantics handles communication. Consider a single client $P_{\mathbf{c}}$ in parallel with three server-side processes:

$$\begin{array}{llll} P_{\mathbf{c}} & \mid P_{\mathbf{s}} & \mid P_{\mathbf{w_1}} & \mid P_{\mathbf{w_2}} \\ = s[\mathbf{c}][\mathbf{s}] \oplus \mathsf{req}\langle 42\rangle \,.\, P_{\mathbf{c}}{}' & \mid !s[\mathbf{s}][\boldsymbol{\alpha}] \,\&\, \mathsf{req}(x) \,.\, P_{\mathbf{s}}{}' & \mid P_{\mathbf{w_1}} & \mid P_{\mathbf{w_2}} \end{array}$$

Using [R-!C$_2$], the client and server reduce. The reduction advances the client to its continuation $P_{\mathbf{c}}{}'$, and pulls out a copy of the server's continuation as a new process. It is key to note that $\boldsymbol{\alpha}$ is acting as a binder in $P_{\mathbf{s}}$, therefore, in the continuation we observe a role variable substitution:

$$\to P_{\mathbf{c}}{}' \mid P_{\mathbf{s}} \mid \left(\sum_{i=1}^{2} s[\mathbf{s}][\mathbf{w_i}] \oplus \mathsf{fw}\langle x, \boldsymbol{\alpha}\rangle \,.\, P_{\mathbf{s_i}}{}''\right) \{42/x\}\{\mathbf{c}/\boldsymbol{\alpha}\} \mid P_{\mathbf{w_1}} \mid P_{\mathbf{w_2}}$$

$$= P_{\mathbf{c}}{}' \mid P_{\mathbf{s}} \mid \sum_{i=1}^{2} s[\mathbf{s}][\mathbf{w_i}] \oplus \mathsf{fw}\langle 42, \mathbf{c}\rangle \,.\, \left(P_{\mathbf{s_i}}{}'' \{42/x\}\{\mathbf{c}/\boldsymbol{\alpha}\}\right) \mid P_{\mathbf{w_1}} \mid P_{\mathbf{w_2}}$$

$$P_c := s[c][s] \oplus \mathsf{req}\langle 42\rangle \ . \ s[c][s] \ \& \ \mathsf{wrk}(\omega) \ . \ s[c][\omega] \ \& \ \mathsf{ans}(z) \ . \ \mathbf{0}$$

$$P_s := !s[s][\alpha] \ \& \ \mathsf{req}(x) \ . \ \sum_{i=1}^{2} s[s][w_i] \oplus \mathsf{fw}\langle x, \alpha\rangle \ . \ s[s][\alpha] \oplus \mathsf{wrk}\langle w_i\rangle \ . \ \mathbf{0}$$

$$P_{w_i} := !s[w_i][s] \ \& \ \mathsf{fw}(y, \gamma) \ . \ s[w_i][\gamma] \oplus \mathsf{ans}\langle\text{“life”}\rangle \ . \ \mathbf{0}$$

Fig. 3. Process definitions for load balancer example

The spawned server process will then non-deterministically choose a worker to send to via rule [R-+]; suppose w_1 is picked.

$$\rightarrow P_c{}' \mid P_s \mid s[s][w_1] \oplus \mathsf{fw}\langle 42, c\rangle \ . \ P_{s_1}{}'' \mid !s[w_1][s] \ \& \ \mathsf{fw}(y, \gamma) \ . \ P_{w_1}{}' \mid P_{w_2}$$

Communication is now possible between the spawned server process and worker w_1, using rule [R-!C$_1$]. As before, this communication advances the sender process and pulls out a copy of the worker's continuation:

$$\rightarrow P_c{}' \mid P_s \mid P_{s_1}{}'' \mid P_{w_1} \mid P_{w_1}{}' \{42/y\}\{c/\gamma\} \mid P_{w_2}$$

The client can now learn which worker was chosen, terminating the spawned server process, then the answer is exchanged between the worker and client:

$$\begin{aligned}
&\rightarrow s[c][w_1] \ \& \ \mathsf{ans}(z) \ . \ \mathbf{0} \mid P_s \mid \mathbf{0} \mid P_{w_1} \mid s[w_1][c] \oplus \mathsf{ans}\langle\text{“life”}\rangle \ . \ \mathbf{0} \mid P_{w_2} \\
&\rightarrow \mathbf{0} \mid P_s \mid \mathbf{0} \mid P_{w_1} \mid \mathbf{0} \mid P_{w_2} \\
&\equiv P_s \mid P_{w_1} \mid P_{w_2}
\end{aligned}$$

2.2 Types

Figure 4 shows the syntax of MPST! types.

Syntax of types. Session types S type communication endpoints. They consist of branching types $S^{\&}$, *replicated* branching types $!(S^{\&})$, selection types $S^{\oplus}$, recursive types $\mu \mathrm{t}.S$ and variables t, and the **end** type.

Branching session types have the form $\rho \&_{i \in I} \mathsf{m}_i(\widetilde{T_i}).S_i$ indicating that role ρ offers a choice of message labels m_i with payload types $\widetilde{T_i}$ and continuations S_i. Similarly, selection session types $\oplus_{i \in I} \rho_i \ \mathsf{m}_i(\widetilde{T_i}).S_i$ indicate an internal choice towards one of a set of roles ρ_i, with a message label, given payload types and continues as the corresponding continuation type. A *replicated* branching type $!\rho \&_{i \in I} \mathsf{m}_i(\widetilde{T_i}).S_i$ types a replicated channel. As with processes, role ρ can either be a concrete role, or a role variable in *binding* position.

Our type system supports *singleton* types for roles: role ρ has singleton type ρ, used to pattern-match specific roles in payloads. *Static* types T are used for protocol specification, whereas *runtime* types U are used by the type semantics to include a notion of *parallel composition* at type level: originally introduced by Le Brun and Dardha [4], a type $U_1 \mid U_2$ allows a channel to be associated with multiple "active" session types.

$$
\begin{array}{rl}
\textit{Session types} & S ::= S^{\&} \mid !(S^{\&}) \mid S^{\oplus} \mid \mu \mathrm{t}.S \mid \mathrm{t} \mid \mathbf{end} \\
\textit{Branching/Selection Types} & S^{\&} ::= \boldsymbol{\rho}\&_{i \in I}\, \mathsf{m}_i(\widetilde{T}_i).S_i \qquad S^{\oplus} ::= \oplus_{i \in I} \boldsymbol{\rho_i}\, \mathsf{m}_i(\widetilde{T}_i).S_i \\
\textit{Role singletons} & \boldsymbol{\rho} ::= \boldsymbol{q} \mid \boldsymbol{\alpha} \\
\textit{Static types} & T ::= S \mid \boldsymbol{\rho} \\
\textit{Runtime types} & U ::= S \mid (U_1 \mid U_2)
\end{array}
$$

Fig. 4. Syntax of Types

We assume that branching and selection types include a non-empty set of messages with pairwise distinct message names m_i (per role for $\oplus$). We further take an *equi-recursive* view of types, identifying a recursive type with its unfolding (i.e., $\mu \mathrm{t}.S = S\{\mu \mathrm{t}.S/\mathrm{t}\}$) and require that recursion variables are guarded.

Definition 2 (Subtyping). *The* ***subtyping relation*** $\leqslant$ *is co-inductively defined on types by the following inference rules:*

$$
\frac{(\widetilde{T}_i \leqslant \widetilde{T'_i})_{i \in I} \qquad (S_i \leqslant S'_i)_{i \in I}}{\boldsymbol{\rho}\&_{i \in I}\, \mathsf{m}_i(\widetilde{T}_i).S_i \leqslant \boldsymbol{\rho}\&_{i \in I \cup J}\, \mathsf{m}_i(\widetilde{T'_i}).S'_i}
\qquad
\frac{(\widetilde{T}_i \geqslant \widetilde{T'_i})_{i \in I} \qquad (S_i \leqslant S'_i)_{i \in I}}{\oplus_{i \in I \cup J} \boldsymbol{\rho_i}\, \mathsf{m}_i(\widetilde{T}_i).S_i \leqslant \oplus_{i \in I}\, \boldsymbol{\rho_i}\, \mathsf{m}_i(\widetilde{T'_i}).S'_i}
$$

$$
\frac{S_1^{\&} \leqslant S_2^{\&}}{!S_1^{\&} \leqslant !S_2^{\&}}
\qquad
\frac{U \leqslant U' \qquad S \leqslant S'}{U \mid S \leqslant U' \mid S'}
\qquad
\frac{S\{\mu \mathrm{t}.S/\mathrm{t}\} \leqslant S'}{\mu \mathrm{t}.S \leqslant S'}
\qquad
\frac{S \leqslant S'\{\mu \mathrm{t}.S'/\mathrm{t}\}}{S \leqslant \mu \mathrm{t}.S'}
\qquad
\frac{}{T \leqslant T}
$$

Our definition of subtyping (Definition 2) is mostly standard. We adopt the convention of smaller types being ones with less external choice and more internal choice (*à la* Gay and Hole [13]). Subtyping of replication is based on regular branching; and parallel types are related iff their session types are subtypes. Subtypes are related up-to their recursive unfolding, and subtyping is *reflexive*.

Definition 3 (Type Congruence). Type congruence *allows us to treat parallel runtime types as commutative and associative with identity element* $\mathbf{end}$.

$$
U_1 \mid U_2 \equiv U_2 \mid U_1 \qquad (U_1 \mid U_2) \mid U_3 \equiv U_1 \mid (U_2 \mid U_3) \qquad U \mid \mathbf{end} \equiv U
$$

Figure 5 shows the definition of typing contexts and their operations. Context Θ is used to type recursive process definitions, mapping process variables X to tuples of parameter types. Context Γ maps channels to types, and role variable singletons. Context composition Γ, Γ' is defined iff Γ and Γ' have disjoint domains. We lift subtyping and type congruence to contexts in the usual way.

As inspired by (e.g.) [30], linearity is enforced through the use of a *context split* operation $\Gamma = \Gamma_1 \cdot \Gamma_2$ that splits a context Γ into two environments Γ_1 and Γ_2. These environments may share variables with unrestricted types and role variables. Additionally, a channel c with runtime type $U_1 \mid U_2$ may be split such that Γ_1 contains $c : U_1$ and Γ_2 contains $c : U_2$; this allows us to type endpoints used by different replicated processes. The inverse operation is *context addition*

Typing contexts

$$\Theta ::= \emptyset \mid \Theta, X : \widetilde{S} \qquad \Gamma ::= \emptyset \mid \Gamma, c : U \mid \Gamma, \boldsymbol{\alpha} : \boldsymbol{\alpha}$$

Context splitting $\boxed{\Gamma = \Gamma_1 \cdot \Gamma_2}$

$$\frac{}{\emptyset = \emptyset \cdot \emptyset} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, c : U = \Gamma_1, c : U \cdot \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, c : U = \Gamma_1 \cdot \Gamma_2, c : U}$$

$$\frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, \boldsymbol{\alpha} : \boldsymbol{\alpha} = \Gamma_1, \boldsymbol{\alpha} : \boldsymbol{\alpha} \cdot \Gamma_2, \boldsymbol{\alpha} : \boldsymbol{\alpha}} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, c : U_1 \mid U_2 = \Gamma_1, c : U_1 \cdot \Gamma_2, c : U_2}$$

Context addition $\boxed{\Gamma_1 + \Gamma_2 = \Gamma}$

$$\frac{}{\Gamma + \emptyset = \Gamma} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma \qquad a \notin \mathsf{dom}(\Gamma_2)}{\Gamma_1, a : T + \Gamma_2 = \Gamma, a : T} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma \qquad a \notin \mathsf{dom}(\Gamma_1)}{\Gamma_1 + \Gamma_2, a : T = \Gamma, a : T}$$

$$\frac{\Gamma_1 + \Gamma_2 = \Gamma}{\Gamma_1, \boldsymbol{\alpha} : \boldsymbol{\alpha} + \Gamma_2, \boldsymbol{\alpha} : \boldsymbol{\alpha} = \Gamma, \boldsymbol{\alpha} : \boldsymbol{\alpha}} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma}{\Gamma_1, c : U_1 + \Gamma_2, c : U_2 = \Gamma, c : U_1 \mid U_2}$$

Context role insertion $\boxed{\Gamma \hookleftarrow \boldsymbol{\rho}}$

$$\Gamma \hookleftarrow \boldsymbol{q} = \Gamma \qquad \Gamma \hookleftarrow \boldsymbol{\alpha} = \Gamma + \boldsymbol{\alpha} : \boldsymbol{\alpha}$$

Fig. 5. Typing contexts and context operations.

$\Gamma_1 + \Gamma_2 = \Gamma$ that combines Γ_1 and Γ_2 into an environment Γ; again roles may be shared across environments. In the case that we have $\Gamma_1, c : U_1 + \Gamma_2, c : U_2$ (i.e., a linear variable used in two different contexts), context addition results in c having the combined runtime type $U_1 \mid U_2$. The *role insertion* operation $\Gamma \hookleftarrow \boldsymbol{\rho}$ is used when typing replicated receives and extends a context with a mapping $\boldsymbol{\alpha} : \boldsymbol{\alpha}$ in the case that $\boldsymbol{\rho}$ is a role variable, and leaves Γ unchanged otherwise.

Before presenting the typing rules, we introduce the notion of a *session protocol*, mapping role names to session types. The $\mathsf{assoc}_s(\Psi)$ function associates a session protocol Ψ to a concrete session s to form a typing context.

Definition 4 (Session Protocol). *A session protocol Ψ is attached to a session through the restriction operation in the form of $(\nu s : \Psi)\ P$, dictating the protocol for s in P. A protocol Ψ is a partial mapping from role to session type, given as:*

$$\Psi ::= \emptyset \mid \Psi, \boldsymbol{q} : S$$

A protocol Ψ is well-formed iff it does not contain parallel types. We obtain a typing context by associating roles in a protocol with a session name. Formally:

$$\mathsf{assoc}_s(\boldsymbol{q} : S, \Psi) := s[\boldsymbol{q}] : S, \mathsf{assoc}_s(\Psi) \qquad \mathsf{assoc}_s(\emptyset) := \emptyset$$

Typing Rules for Values and Recursive Definitions $\boxed{\Gamma \vdash V : T}$ $\boxed{\Theta \vdash X : \widetilde{S}}$

$$\text{T-Wkn}\ \dfrac{\Gamma_1 + \Gamma_2 = \Gamma \qquad \Gamma_1 \vdash V : T \quad \mathsf{end}(\Gamma_2)}{\Gamma \vdash V : T} \qquad \text{T-}\boldsymbol{q}\ \dfrac{}{\emptyset \vdash \boldsymbol{q}{:}\boldsymbol{q}} \qquad \text{T-Sub}\ \dfrac{S \leqslant S'}{c{:}S \vdash c{:}S'} \qquad \text{T-}\boldsymbol{\alpha}\ \dfrac{}{\boldsymbol{\alpha}{:}\boldsymbol{\alpha} \vdash \boldsymbol{\alpha}{:}\boldsymbol{\alpha}} \qquad \text{T-X}\ \dfrac{\Theta(\mathrm{X}){=}(S_i)_{i \in 1..n}}{\Theta \vdash \mathrm{X}{:}(S_i)_{i \in 1..n}}$$

Typing Rules for Processes $\boxed{\Theta ; \Gamma \vdash P}$

$$\text{T-}\mathbf{0}\ \dfrac{\mathsf{end}(\Gamma)}{\Theta ; \Gamma \vdash 0} \qquad \text{T-}|\ \dfrac{\Theta ; \Gamma_1 \vdash P_1 \quad \Theta ; \Gamma_2 \vdash P_2}{\Theta ; \Gamma_1 \cdot \Gamma_2 \vdash P_1 \mid P_2} \qquad \text{T-}+\ \dfrac{(\Theta ; \Gamma \vdash c_i[\boldsymbol{\rho_i}] \oplus \mathsf{m}_i\langle \widetilde{d_i} \rangle \ . \ P_i)_{i \in I}}{\Theta ; \Gamma \vdash \sum_{i \in I} c_i[\boldsymbol{\rho_i}] \oplus \mathsf{m}_i\langle \widetilde{d_i} \rangle \ . \ P_i}$$

$$\text{T-!}\ \dfrac{\Gamma_! \vdash c : !\boldsymbol{\rho} \&_{i \in I} \mathsf{m}_i(\widetilde{T_i}).S'_i \quad \mathsf{end}(\Gamma) \qquad (\Theta ; \Gamma + c : S'_i + \widetilde{b_i} : \widetilde{T_i} \hookleftarrow \boldsymbol{\rho} \vdash P_i)_{i \in I}}{\Theta ; \Gamma \cdot \Gamma_! \vdash !c[\boldsymbol{\rho}] \ \&_{i \in I} \mathsf{m}_i(\widetilde{b_i}) \ . \ P_i}$$

$$\text{T-}\&\ \dfrac{\Gamma_\& \vdash c : \boldsymbol{\rho} \&_{i \in I} \mathsf{m}_i(\widetilde{T_i}).S'_i \quad \Gamma_r \vdash \boldsymbol{\rho} : \boldsymbol{\rho} \qquad (\Theta ; \Gamma + c : S'_i + \widetilde{b_i} : \widetilde{T_i} \vdash P_i)_{i \in I}}{\Theta ; \Gamma \cdot \Gamma_\& \cdot \Gamma_r \vdash c[\boldsymbol{\rho}] \ \&_{i \in I} \mathsf{m}_i(\widetilde{b_i}) \ . \ P_i}$$

$$\text{T-}\nu\ \dfrac{\varphi(\mathsf{assoc}_s(\Psi)) \qquad \varphi \text{ is a } \textit{safety} \text{ property} \qquad s \notin \Gamma \quad \Theta ; \Gamma + \mathsf{assoc}_s(\Psi) \vdash P}{\Theta ; \Gamma \vdash (\nu s : \Psi) \ P}$$

$$\text{T-}\oplus\ \dfrac{\Gamma_\oplus \vdash c : \boldsymbol{\rho} \oplus \mathsf{m}(\widetilde{T}).S' \quad \Gamma_r \vdash \boldsymbol{\rho} : \boldsymbol{\rho} \qquad (\Gamma_i \vdash V_i : T_i)_{i \in 1..n} \quad \Theta ; \Gamma + c : S' \vdash P}{\Theta ; \Gamma \cdot \Gamma_\oplus \cdot \Gamma_r (\cdot \, \Gamma_i)_{i \in 1..n} \vdash c[\boldsymbol{\rho}] \oplus \mathsf{m}\langle (V_i)_{i \in 1..n} \rangle \, . \, P}$$

$$\text{T-Def}\ \dfrac{\Theta, \mathrm{X} : \widetilde{S} ; \Gamma, \widetilde{x : S} \vdash P \quad \mathsf{end}(\Gamma) \qquad \Theta, \mathrm{X} : \widetilde{S} ; \Gamma' \vdash Q}{\Theta ; \Gamma \cdot \Gamma' \vdash \mathsf{def}\ \mathrm{X}(\widetilde{x : S}) = P \ \mathsf{in}\ Q} \qquad \text{T-Call}\ \dfrac{\Theta \vdash \mathrm{X} : (S_i)_{i \in 1..n} \quad \mathsf{end}(\Gamma_0) \qquad \forall i \in 1..n : \Gamma_i \vdash c_i : S_i}{\Theta ; \Gamma_0 (\cdot \, \Gamma_i)_{i \in 1..n} \vdash \mathrm{X}\langle (c_i)_{i \in 1..n} \rangle}$$

Fig. 6. Typing rules.

Next, we introduce predicate end on type contexts. This ensures that an environment is *end-typed*: that is, its session types are congruent to type **end**.

Definition 5 (End-typed environment). *A context is* end-typed*, written* $\mathsf{end}(\Gamma)$*, iff it only maps channels to session type* **end**.

$$\dfrac{}{\mathsf{end}(\emptyset)} \qquad \dfrac{\mathsf{end}(\Gamma)}{\mathsf{end}(c : \mathbf{end}, \Gamma)} \qquad \dfrac{\mathsf{end}(\Gamma)}{\mathsf{end}(\boldsymbol{\alpha} : \boldsymbol{\alpha}, \Gamma)} \qquad \dfrac{\Gamma \equiv \Gamma' \quad \mathsf{end}(\Gamma)}{\mathsf{end}(\Gamma')}$$

Figure 6 shows the typing rules for MPST!. There are three judgements: the value typing judgement $\Gamma \vdash V : T$ assigns type T to value V under context Γ and consists of four rules. Rule [T-Wkn] allows weakening: if value V has type T under some environment Γ_1, and we have an end-typed environment Γ_2 such that $\Gamma_1 + \Gamma_2 = \Gamma$, then the rule allows us to conclude that V has type T under Γ. Rule [T-$\boldsymbol{q}$] types concrete roles as singleton types under the empty environment, whereas [T-$\boldsymbol{\alpha}$] types a role variable provided that it is contained within the type environment. Finally, rule [T-Sub] types a linear variable, allowing for subtyping. Judgement $\Theta \vdash X : \widetilde{S}$ simply looks up process variable X in process environment Θ, returning its parameter types; this is achieved by rule [T-X].

The final judgement $\Theta\,;\Gamma \vdash P$ states that process P is typable under recursion environment Θ and type environment Γ. Rule [T-**0**] types the inactive process under an end-typed environment. Rule [T-|] types parallel processes under a split environment. Rule [T-+] types an output-directed choice; since this operation uses branching control flow, each choice must be typable under the same environment.

Rules [T-!] and [T-&] type replicated and non-replicated receives respectively. In both cases the rules check that the channel has a receiving session type. Both rules check that each branch is typable by extending a common typing context with the variables bound by the receives, along with the continuation type for the session channel. In the case of a replicated receive there are two differences: first, the context used to type each branch must be end-typed (in order to avoid duplicating linear resources). Second, since the role $\boldsymbol{\rho}$ may be a *binding* occurrence of a role variable $\boldsymbol{\alpha}$, the context used to type each branch must be extended with the role variable (if applicable) using the insertion operator.

Rule [T-ν] types a session name restriction $(\nu s : \Psi)P$. As is standard in generalised MPST [26], the session protocol must satisfy a *safety property* φ. We discuss specifics of this property in Section 2.3, and how it is used in Section 2.4. Informally, the property ensures that all process communication is "compatible", and is the weakest property required to prove subject reduction. We then prove our metatheory parametric on the *largest* safety property, allowing it to be customised to verify more precise properties (e.g., to ensure deadlock-freedom or termination), based on the requirements of a specific protocol.

Rule [T-$\oplus$] types an output if the sending channel can be mapped to a selection type with payload types that match the values being sent. The rule ensures that role $\boldsymbol{\rho}$ is either a value, or it exists in the type context—*i.e.,* messages cannot be sent to unbound role variables. The selection type continuation should be used, along with the common context, to type the continuation process.

Lastly, rules [T-Def] and [T-Call] type recursive processes. The former populates the recursion environment whilst ensuring that process declarations are well typed under the variables they bind. The latter checks that types of values used in a process call match what was specified in the declaration.

Example 4 (Load Balancer: Type Checking). By introducing a name restriction for session s that includes the types from Example 2, we can type the processes from Example 3 under empty typing contexts:

$$\emptyset; \emptyset \vdash (\nu s : \{\boldsymbol{s} : S_{\boldsymbol{s}}, \boldsymbol{w_1} : S_{\boldsymbol{w_1}}, \boldsymbol{w_2} : S_{\boldsymbol{w_2}}, \boldsymbol{c} : S_{\boldsymbol{c}}\})\ P_{\boldsymbol{s}} \mid P_{\boldsymbol{w_1}} \mid P_{\boldsymbol{w_2}} \mid P_{\boldsymbol{c}}$$

2.3 Type Semantics

To reason about the interactions between session types, following Scalas & Yoshida [26], we endow typing contexts Γ with LTS semantics as shown in Figure 7. Each action γ denotes an *output*, *input*, and *synchronising communication* respectively. *Context reduction* $\Gamma \rightarrow \Gamma'$ is defined iff $\Gamma \xrightarrow{s:\boldsymbol{p},\boldsymbol{q}:\mathsf{m}} \Gamma'$ for some

$$\text{Actions} \quad \gamma ::= s{:}\boldsymbol{q}{\oplus}\boldsymbol{r}{:}\mathsf{m}(\widetilde{T}) \;\big|\; s{:}\boldsymbol{q}\&\boldsymbol{\rho}{:}\mathsf{m}(\widetilde{T}) \;\big|\; s{:}\boldsymbol{q},\boldsymbol{r}{:}\mathsf{m}$$

Context Reduction $\boxed{\Gamma \to \Gamma'}$

$$\Gamma\text{-}\& \quad \frac{S = \boldsymbol{q}\&_{i\in I}\, \mathsf{m}_i(\widetilde{T_i}).S_i' \qquad k \in I}{s[\boldsymbol{p}] : S \xrightarrow{s:\boldsymbol{p}\&\boldsymbol{q}:\mathsf{m}_k(\widetilde{T_k})} s[\boldsymbol{p}] : S_k'}
\qquad
\Gamma\text{-}\oplus \quad \frac{S = \oplus_{i\in I}\, \boldsymbol{q_i}\mathsf{m}_i(\widetilde{T_i}).S_i' \qquad k \in I}{s[\boldsymbol{p}] : S \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q_k}:\mathsf{m}_k(\widetilde{T_k})} s[\boldsymbol{p}] : S_k'}
\qquad
\Gamma\text{-}\boldsymbol{\rho} \quad \frac{\Gamma \xrightarrow{\gamma} \Gamma'}{\Gamma, \boldsymbol{\rho} : \boldsymbol{\rho} \xrightarrow{\gamma} \Gamma', \boldsymbol{\rho} : \boldsymbol{\rho}}$$

$$\Gamma\text{-}! \quad \frac{R = !\boldsymbol{\rho}\&_{i\in I}\, \mathsf{m}_i(\widetilde{T_i}).S_i \qquad k \in I}{s[\boldsymbol{q}] : R \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{\rho}:\mathsf{m}_k(\widetilde{T_k})} s[\boldsymbol{q}] : R \mid S_k}
\qquad
\Gamma\text{-}\textsc{Cong} \quad \frac{\Gamma \xrightarrow{\gamma} \Gamma'}{\Gamma \cdot \Gamma'' \xrightarrow{\gamma} \Gamma' + \Gamma''}
\qquad
\Gamma\text{-}\mu \quad \frac{\Gamma \cdot c : S\{\mu \mathrm{t}.S/\mathrm{t}\} \xrightarrow{\gamma} \Gamma'}{\Gamma \cdot c : \mu \mathrm{t}.S \xrightarrow{\gamma} \Gamma'}$$

$$\Gamma\text{-}\textsc{Com}_1 \quad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}(\widetilde{S},\widetilde{\boldsymbol{r}})} \Gamma_1' \qquad \Gamma_2 \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{p}:\mathsf{m}(\widetilde{S}',\widetilde{\boldsymbol{\alpha}})} \Gamma_2' \qquad \widetilde{S} \leqslant \widetilde{S}'}{\Gamma \xrightarrow{s:\boldsymbol{p},\boldsymbol{q}:\mathsf{m}} \Gamma_1' + \Gamma_2'\{\widetilde{\boldsymbol{r}}/\widetilde{\boldsymbol{\alpha}}\}}$$

$$\Gamma\text{-}\textsc{Com}_2 \quad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}(\widetilde{S},\widetilde{\boldsymbol{r}})} \Gamma_1' \qquad \Gamma_2 \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{\alpha_0}:\mathsf{m}(\widetilde{S}',\widetilde{\boldsymbol{\alpha}})} \Gamma_2' \qquad \widetilde{S} \leqslant \widetilde{S}'}{\Gamma \xrightarrow{s:\boldsymbol{p},\boldsymbol{q}:\mathsf{m}} \Gamma_1' + \Gamma_2'\{\boldsymbol{p}/\boldsymbol{\alpha_0}\}\{\widetilde{\boldsymbol{r}}/\widetilde{\boldsymbol{\alpha}}\}}$$

Fig. 7. Type semantics.

$s, \boldsymbol{p}, \boldsymbol{q}, \mathsf{m}$, and we write $\to^*$ for the transitive and reflexive closure of $\to$. We write $\Gamma\to$ iff $\Gamma\to\Gamma'$ for some Γ'.

Transitions [Γ-&] and [Γ-$\oplus$] are standard: a context can fire an input label (resp. output label) matching any of the labels in the top-level branch type (resp. selection type). This transitions the entry to the continuation of the chosen type.

Transition [Γ-!] models the receipt of a message by a replicated input. The two main differences to the linear receive are: *(i)* the role $\boldsymbol{\rho}$ used in the transition label is allowed to be a role variable name; and *(ii)* firing an input does not advance the type, but instead pulls out a copy of the continuation and places it in parallel. Role $\boldsymbol{\rho}$ is considered bound in R and its continuations, but is *free* in pulled out copies of the continuations composed in parallel (S_k).

Transition rules [Γ-$\boldsymbol{\rho}$], [Γ-Cong], [Γ-μ] allow contexts to reduce under a larger context, or when types are guarded by recursive binders. Concretely, [Γ-$\boldsymbol{\rho}$] allows transitions to ignore role singletons; [Γ-Cong] allows a context to perform a transition when split from a larger context (the transition result must be added back in); and [Γ-μ] allows recursive binders to be treated equal to their unfolding.

Transitions [Γ-Com$_1$] and [Γ-Com$_2$] model type-level communication; for simplicity and without loss of generality we assume a convention wherein session-typed payloads precede role-typed payloads. Both rules state that if a context can be split such that one part fires an output label, and the other fires an input with matching roles, message label and payloads, then the entire context can transition via a *communication* action. We note that payloads will consist of either session types or role singletons. For the former, sender payloads must be subtypes of the receiver payloads; for the latter, *role substitution* occurs after communication. [Γ-Com$_2$] caters for universal receives, where the input label consists of a role variable in binding position—this is reflected in the role substitution. We say that a context *reduces* iff it can transition via communication actions.

Example 5 (Load Balancer: Context Reduction). We now use the protocol from Example 4 to demonstrate context reduction in action. Initially, the only communication action possible is between the client and server, via [Γ-Com$_2$].

$$
\begin{aligned}
& s[c]: S_c,\ s[s]: S_s,\ s[w_1]: S_{w_1},\ s[w_2]: S_{w_2} \\
= \ & \\
& (s[c]: S_c \cdot s[s]: S_s) \cdot (s[w_1]: S_{w_1},\ s[w_2]: S_{w_2}) \\
\to \ & \\
& s[c]: s\&\mathsf{wrk}(\omega).\ \omega\&\mathsf{ans}(\mathtt{str}) + \left(s[s]: S_s \;\middle|\; \oplus \begin{cases} w_1\mathsf{fw}(\mathtt{int}, \alpha).\ \alpha\oplus\mathsf{wrk}(w_1) \\ w_2\mathsf{fw}(\mathtt{int}, \alpha).\ \alpha\oplus\mathsf{wrk}(w_2) \end{cases}\right)\{c/\alpha\} \\
& + s[w_1]: S_{w_1},\ s[w_2]: S_{w_2} \\
= \ & \\
& s[c]: s\&\mathsf{wrk}(\omega).\ \omega\&\mathsf{ans}(\mathtt{str}), \\
& s[s]: S_s \;\middle|\; \oplus \begin{cases} w_1\mathsf{fw}(\mathtt{int}, c).\ c\oplus\mathsf{wrk}(w_1) \\ w_2\mathsf{fw}(\mathtt{int}, c).\ c\oplus\mathsf{wrk}(w_2) \end{cases},\ s[w_1]: S_{w_1},\ s[w_2]: S_{w_2}
\end{aligned}
$$

It is key to note the role substitution for the type of $s[s]$ above; specifically, how the substitution affects the parallel type extracted through communication, but does not affect the replicated type S_s. From here, there are multiple reduction paths, but let us observe the reduction path in which s communicates with w_1.

$$
\begin{aligned}
= \ & \\
& s[c]: s\&\mathsf{wrk}(\omega).\ \omega\&\mathsf{ans}(\mathtt{str}),\ s[w_2]: S_{w_2},\ s[s]: S_s \\
& \cdot \left(s[s]: \oplus \begin{cases} w_1\mathsf{fw}(\mathtt{int}, c).\ c\oplus\mathsf{wrk}(w_1) \\ w_2\mathsf{fw}(\mathtt{int}, c).\ c\oplus\mathsf{wrk}(w_2) \end{cases} \cdot s[w_1]: S_{w_1}\right) \\
\to \ & \\
& s[c]: s\&\mathsf{wrk}(\omega).\ \omega\&\mathsf{ans}(\mathtt{str}),\ s[w_2]: S_{w_2},\ s[s]: S_s \\
& + (s[s]: c\oplus\mathsf{wrk}(w_1) + (s[w_1]: S_{w_1} \mid \gamma\oplus\mathsf{ans}(\mathtt{str}))\{c/\gamma\}) \\
= \ & \\
& s[c]: s\&\mathsf{wrk}(\omega).\ \omega\&\mathsf{ans}(\mathtt{str}),\ s[w_2]: S_{w_2}, \\
& s[s]: S_s \mid c\oplus\mathsf{wrk}(w_1),\ s[w_1]: S_{w_1} \mid c\oplus\mathsf{ans}(\mathtt{str})
\end{aligned}
$$

Note how reduction is possible because the context split allows the server's parallel type to be extracted into its own context. Then, reduction occurs via [Γ-Cong] and [Γ-Com$_1$]. From here, the context reduces in a similar fashion: first the server communicates a role with the client; followed by the final communication between the client and worker.

$$
\begin{aligned}
\to \ & \\
& s[c]: w_1\&\mathsf{ans}(\mathtt{str}),\ s[s]: S_s \mid \mathbf{end},\ s[w_2]: S_{w_2},\ s[w_1]: S_{w_1} \mid c\oplus\mathsf{ans}(\mathtt{str}) \\
\to \ & \\
& s[c]: \mathbf{end},\ s[s]: S_s \mid \mathbf{end},\ s[w_2]: S_{w_2},\ s[w_1]: S_{w_1} \mid \mathbf{end} \\
\equiv \ & \\
& s[c]: \mathbf{end},\ s[s]: S_s,\ s[w_2]: S_{w_2},\ s[w_1]: S_{w_1}
\end{aligned}
$$

Safety. In order to type a session restriction, rule [T-ν] in Figure 6 requires that the session's protocol of the session must obey a *safety property* φ. Informally, a safety property requires that processes exchange values of compatible types and that a sender only ever selects available branches. Safety is the weakest typing context property required in order to prove subject reduction.

Definition 6 (Safety). *φ is a safety property on type environment Γ iff:*

S-$\oplus\&$

$$\varphi\left(\Gamma \cdot s[\boldsymbol{p}] : \oplus_{i\in I}\, \boldsymbol{\rho_i}\, \mathsf{m}_i(\widetilde{S_i}, \widetilde{\boldsymbol{r_i}}).S'_i \cdot s[\boldsymbol{q}] : \boldsymbol{p}\ \&_{j\in J}\, \mathsf{m}_j(\widetilde{S''_j}, \widetilde{\boldsymbol{\alpha_j}}).S'''_j\right)$$
and $\exists K \subseteq I$ s.t. $\forall k \in K : \boldsymbol{\rho_k} = \boldsymbol{q}$
implies $K \subseteq J$ and $\forall i \in K : \widetilde{S_i} \leqslant \widetilde{S''_i}$ and $|\widetilde{\boldsymbol{r_i}}| = |\widetilde{\boldsymbol{\alpha_i}}|$

S-$!\oplus\&$

$$\varphi\left(\Gamma \cdot s[\boldsymbol{p}] : \oplus_{i\in I}\, \boldsymbol{\rho_i}\, \mathsf{m}_i(\widetilde{S_i}, \widetilde{\boldsymbol{r_i}}).S'_i \cdot s[\boldsymbol{q}] : !\boldsymbol{\rho_0}\ \&_{j\in J}\, \mathsf{m}_j(\widetilde{S''_j}, \widetilde{\boldsymbol{\alpha_j}}).S'''_j\right)$$
and $\exists K \subseteq I$ s.t. $\forall k \in K : \boldsymbol{\rho_k} = \boldsymbol{q}$ and $\boldsymbol{\rho_0}$ is either a variable or $\boldsymbol{p}$
implies $K \subseteq J$ and $\forall i \in K : \widetilde{S_i} \leqslant \widetilde{S''_i}$ and $|\widetilde{\boldsymbol{r_i}}| = |\widetilde{\boldsymbol{\alpha_i}}|$

S-μ

$\varphi(\Gamma \cdot s[\boldsymbol{q}] : \mu\mathrm{t}.S)$ implies $\varphi(\Gamma \cdot s[\boldsymbol{q}] : S\{\mu\mathrm{t}.S/\mathrm{t}\})$

S-$\boldsymbol{\alpha}$

$\varphi(\Gamma)$ implies $\mathsf{frv}(\Gamma) = \emptyset$

S-$\rightarrow$

$\varphi(\Gamma)$ and $\Gamma \rightarrow \Gamma'$ implies $\varphi(\Gamma')$

A property φ is considered *safe* iff it conforms to all conditions specified in Definition 6. Conditions [S-$\oplus\&$] and [S-$!\oplus\&$] are concerned with communication. They state that if two roles in the same session have an output and input type respectively which point at each other, then: *(i)* they should have at least one common message label; *(ii)* for each common label, their payloads should be equal in length; and *(iii)* for each common label, all session types in the payload of the sender should be subtypes of what is expected by the receiver.

Condition [S-μ] requires safety to hold after the unfolding of recursive binders; [S-$\boldsymbol{\alpha}$] requires all role variables used in a context to be bound by that same context; and [S-$\rightarrow$] requires safety to hold after context reduction.

Users of the type system can re-instantiate φ with custom properties (*e.g.*, *termination*), as long as the property used meets the safety requirements.

2.4 Metatheory

Unlike most session type theories, *generalised* MPST do not syntactically guarantee any properties on the processes they type. Rather, they provide a framework for verifying runtime properties on the type context, from which process-level properties may be inferred—the benefit being that these properties are undecidable to check on processes, yet decidable in the realm of the type system. Furthermore, its generalised nature allows for fine-tuning based on specific requirements of its applications. Informally, generalisation of the type system works by proving the metatheory parametric of the largest safety property captured by φ in Definition 6; *i.e.,* all theorems proved and presented which involve a type context Γ, assume that $\varphi(\Gamma)$ holds, or "Γ *is safe*". Essentially, φ describes the minimum (and most general) *safety* requirements made for *subject reduction* to hold. This proof technique allows φ to be re-instantiated with more specific properties without having to reprove any of the base metatheory. (We occasionally reference properties other than *safety* in examples.)

2.5 Subject Reduction and Session Fidelity

The main results of a generalised MPST system are *subject reduction* (Theorem 1) and *session fidelity* (Theorem 2). These theorems allow the type system to be used as a framework for verifying custom properties by re-instantiating φ. We note that discussing *how* the generalised type system can be used as a verification framework is *not* the focus of this paper (to this end, we address the interested reader to Scalas and Yoshida [26, Section 5]); rather, we build on generalised MPST theory as a means of investigating the expressiveness of replication and first-class roles in MPST. Hence, the following presents the two theorems—highlighting key differences with what is standard—but we do not demonstrate the verification of runtime properties (which is standard). We give the proofs in Appendix C.

Theorem 1 (Subject Reduction). *If $\Theta\,;\Gamma \vdash P$ with $\varphi(\Gamma)$ and $P \to P'$, then $\exists \Gamma'$ s.t. $\Gamma \to^* \Gamma'$ and $\Theta\,;\Gamma' \vdash P'$ with $\varphi(\Gamma')$.*

Intuitively, subject reduction states that any *safe* and *well-typed* process remains safe and well-typed after process reduction. More formally, the theorem asserts that if a process is typed under a safe context, then the context can match any process reduction to type its continuation whilst retaining safety.

Session fidelity states that if a context can reduce, then a process it types can observe *at least one* of its reductions. By virtue of subtyping, a context is allowed to specify paths which need not be followed by a process it types; the key point is that session fidelity requires that there is *at least one* observable reduction. Using session fidelity, one can prove properties about communication occurring within a single session. It does not, however, provide grounds for showing such properties on interleaved session communication. Hence, as is standard in generalised MPST, we define additional assumptions on processes required for session fidelity to hold.

Definition 7 (Only plays one role). *(The following is a slight adaptation of [26, definition 5.3].) Assuming $\emptyset\,;\Gamma \vdash P$, we say P:*

1. ***has guarded definitions** iff each subterm of P with the form*

$$\mathit{def}\ \mathrm{X}((x_i : S_i)_{i \in 1..n}) = Q\ \mathit{in}\ P'$$

   we have: $\forall i \in 1..n : S_i \not\leqslant \mathbf{end}$ implies a process call $\mathrm{Y}\langle \ldots, x_i, \ldots \rangle$ can only occur in Q as a subterm of a communication action over channel x_i.
2. ***only plays role $\mathbf{p}$ in s, by Γ** iff (i) Item 1 holds for P; (ii) $\mathsf{fv}(P) = \emptyset$; (iii) $\Gamma = \Gamma_0, s[\mathbf{p}] : U$ with $U \not\leqslant \mathbf{end}$ and $\mathsf{end}(\Gamma_0)$; (iv) in all subterms $(\nu s : \Gamma')\ P'$ of P, we have $\mathsf{end}(\Gamma')$.*

The purpose of Item 1 is to prevent processes from infinitely reducing via [R-X] without communicating, and Item 2 identifies a typical application of MPST where a number of processes $P_{\mathbf{p}}$ communicate over a multiparty session s, with each process playing a *single* role. The difference in our definition to the standard is that processes $P_{\mathbf{p}}$ should play a *single* role and not a *unique* role. This is due to the introduction of replication in our language; note how reduction with a replicated process is guaranteed to produce multiple processes playing the same role and is reflected in our definition in condition *(iii)* of Item 2, where a channel is mapped to *runtime type* U, allowing for parallel composition.

Informally, the session fidelity theorem states that, given a safe context that types a process of a particular structure—*i.e.,* one governed by the session fidelity assumptions of Definition 7—then if the context can reduce, the typed process can match at least one reduction. Furthermore, after the process matches the context reduction, it remains within the session fidelity assumption structure.

Theorem 2 (Session Fidelity). *Assume* $\emptyset\,;\Gamma \vdash P$ *with* $\varphi(\Gamma)$ *and* $P \equiv \big|_{\mathbf{p}\in I} P_{\mathbf{p}}$ *where each* $P_{\mathbf{p}}$ *is either* $\mathbf{0}$ *(up-to* $\equiv$*), or only plays role* $\mathbf{p}$ *in* s*. Then,* $\Gamma \rightarrow$ *implies* $\exists \Gamma', P'$ *s.t.* $\Gamma \rightarrow \Gamma'$, $P \rightarrow^* P'$ *and* $\Gamma' \vdash P'$*, where* $P' \equiv \big|_{\mathbf{p}\in I} P_{\mathbf{p}}{}'$ *and each* $P_{\mathbf{p}}{}'$ *is either* $\mathbf{0}$ *(up-to* $\equiv$*), or only plays role* $\mathbf{p}$ *in* s*.*

We now turn our attention to the main focus of this paper, *i.e.,* exploring the expressiveness and decidability of replication and first-class roles in MPST.

3 Expressivity and Decidability

This section discusses, and shows by example, the benefits and limitations of MPST!. Section 3.1 demonstrates the exressivity gained by using replication and first-class roles in MPST, whilst Section 3.2 presents our decidability results.

3.1 Expressivity

We begin by demonstrating a common design pattern used for describing protocols, which we call *services*. We build a number of services to showcase language features, and to describe protocols which—to the best of our knowledge—were previously untypable in any MPST theory. Specifically, using the increased expressiveness of replication and first-class roles, we define types for *binary trees*, the *dining philosophers problem*, and an *auction*. Importantly, all examples shown adhere to the decidability requirements discussed later (*cf.* Section 3.2).

Services. A *service* is a building-block of a protocol, involving some universal receive, with the aim of *offering* a specific interaction. A *client* interacts with a service to achieve the communication pattern it offers. Importantly, services may be clients of other services, promoting a modular design of protocols in MPST!.

Example 6 (Ping). The *ping service* simply responds to a ping with a pong.

$$\boldsymbol{P} : !\boldsymbol{\alpha}\&\mathsf{ping}\,.\,\boldsymbol{\alpha}\oplus\mathsf{pong}\,.\,\mathbf{end}$$

A basic yet useful service is given in Example 6, where role $\boldsymbol{P}$ offers a *ping* service. As a convention, we will use capitals for naming services. We highlight the importance of both replication and free role names in types to be able to design modular components of a protocol—both are integral to designing a sub-protocol agnostic of the larger scope in which it is used.

Context-Free MPST. Context-free session types [28,1,21] are a formalism that replace prefix-style session types with individual actions, along with a sequencing operator **;** with neutral element skip. The goal of this line of work is to express communication protocols that are not possible under tail-recursive session types, given their restriction to regular languages. The classic example is that of communicating a serialised binary tree.

Example 7 (Binary tree in standard context-free STs). Consider a binary tree data type described by the following context-free grammar.

$$\mathsf{tree} ::= (\mathsf{node}, \mathsf{tree}, \mathsf{tree}) \quad | \quad \mathsf{leaf}$$

We could attempt to represent a protocol that serialises this data type as follows:

$$\mu \mathrm{t}. \oplus \{\mathsf{leaf} : \mathsf{skip},\ \mathsf{node} : \mathrm{t}\}$$

However, this type is not precise enough—it does not guarantee that the correct structure of a binary tree is maintained. Work on context-free session types solves this by proposing type systems allowing the following specification:

$$\mu \mathrm{t}. \oplus \{\mathsf{leaf} : \mathsf{skip},\ \mathsf{node} : \mathsf{skip}; \mathrm{t}; \mathrm{t}\}$$

Selecting the node label now guarantees that *two* sub-trees will follow.

The parallel types presented in Section 2.2, although not exposed directly to users, lift expressiveness of types in MPST!. In fact, since replicated branches are *permanently available* (by composing continuation types in parallel), we can simulate the sequencing operator **;** using type-level parallel composition.

Example 8 (Binary Tree Service). Recall the ping service $\boldsymbol{P}$ from Example 6. We build a *binary tree service* $\boldsymbol{T}$ using $\boldsymbol{P}$ as shown below:

$$\boldsymbol{T} : !\boldsymbol{\beta}\&\mathsf{tree}\,.\,\boldsymbol{P}\oplus\mathsf{ping}\,.\,!\boldsymbol{P}\&\mathsf{pong}\,.\,\boldsymbol{\beta}\&\,\{\mathsf{leaf}\,.\,\mathbf{end},\ \mathsf{node}\,.\,\boldsymbol{P}\oplus\mathsf{ping}\,.\,\boldsymbol{P}\oplus\mathsf{ping}\,.\,\mathbf{end}\}$$

The service begins by receiving a request for a tree from a client. It then sends a ping to the ping service, exposing a replicated branch waiting to receive the pong reply. The client is now free to build the binary tree. It is key to note that any node sent to the service will subsequently forward *two* ping requests to $\boldsymbol{P}$. In turn, this communication will pull out two copies of the type continuation $\boldsymbol{\beta}\&\{\mathsf{leaf}\,.\,\mathbf{end},\ \mathsf{node}\,.\,\boldsymbol{P}\oplus\mathsf{ping}\,.\,\boldsymbol{P}\oplus\mathsf{ping}\,.\,\mathbf{end}\}$, forcing the client to maintain the appropriate binary tree structure. For example, if a client $\boldsymbol{t}$ wishes to build a tree consisting of one root node and two leaf nodes, its type would be defined as:

$$\boldsymbol{t} : \boldsymbol{T}\oplus\mathsf{tree}\,.\,\boldsymbol{T}\oplus\mathsf{node}\,.\,\boldsymbol{T}\oplus\mathsf{leaf}\,.\,\boldsymbol{T}\oplus\mathsf{leaf}\,.\,\mathbf{end}$$

The metatheoretic framework can be used to determine that any protocol failing to abide by the binary tree structure will result in a *deadlock*; whilst any safe protocol that obeys the correct structure is *terminating*, *e.g.* the protocol $\{\boldsymbol{t}, \boldsymbol{T}, \boldsymbol{P}\}$.

An obscure limitation of the tree service in Example 8 is that it can only be used by a single client. Consider, for example, two separate clients sending a node message to $\boldsymbol{T}$. Since both tree service types communicate with $\boldsymbol{P}$ to unroll the replicated branch $!\boldsymbol{P}\&\mathsf{pong}$, the protocol becomes non-deterministic in a non-confluent manner and can result in deadlocked behaviour. To resolve this issue, we amend the tree service to accept a payload role which should act as a personal ping service for the client; this guarantees that the tree type is only unrolled by the client that made the initial request.

Example 9 (Multi-Client Binary Tree). We now redesign the binary tree service, this time capable of concurrently building multiple trees for different clients. The key difference here being that the new service, $\boldsymbol{M}$, accepts a role as a payload on the initial request to which it will issue its pings.

$$\boldsymbol{M} : !\boldsymbol{\alpha}\&\mathsf{tree}(\boldsymbol{\beta}) \,.\, \boldsymbol{\beta}\oplus\mathsf{ping} \,.\, !\boldsymbol{\beta}\&\mathsf{pong} \,.\, \boldsymbol{\alpha}\&\,\{\mathsf{leaf} \,.\, \mathbf{end}, \mathsf{node} \,.\, \boldsymbol{\beta}\oplus\mathsf{ping} \,.\, \boldsymbol{\beta}\oplus\mathsf{ping} \,.\, \mathbf{end}\}$$

$$S_{\boldsymbol{p}} = !\boldsymbol{M}\&\mathsf{ping} \,.\, \boldsymbol{M}\oplus\mathsf{pong} \,.\, \mathbf{end}$$

Multiple clients can now issue concurrent requests to the tree service whilst maintaining safety. A sample (terminating) protocol is that of $\{\boldsymbol{t_1}, \boldsymbol{t_2}, \boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{M}\}$, where $\boldsymbol{p_1}, \boldsymbol{p_2} : S_{\boldsymbol{p}}$, and the types for $\boldsymbol{t_1}, \boldsymbol{t_2}$ are given by:

$$\boldsymbol{t_1} : \boldsymbol{M}\oplus\mathsf{tree}(\boldsymbol{p_1}) \,.\, \boldsymbol{M}\oplus\mathsf{node} \,.\, \boldsymbol{M}\oplus\mathsf{leaf} \,.\, \boldsymbol{M}\oplus\mathsf{leaf} \,.\, \mathbf{end}$$
$$\boldsymbol{t_2} : \boldsymbol{M}\oplus\mathsf{tree}(\boldsymbol{p_2}) \,.\, \boldsymbol{M}\oplus\mathsf{node} \,.\, \boldsymbol{M}\oplus\mathsf{leaf} \,.\, \boldsymbol{M}\oplus\mathsf{node} \,.\, \boldsymbol{M}\oplus\mathsf{leaf} \,.\, \boldsymbol{M}\oplus\mathsf{leaf} \,.\, \mathbf{end}$$

Replication vs. Recursion. We have seen that replication and parallel composition increases the expressive power of MPST beyond that of tail-recursion. Naturally, one might ask, "*is recursion still needed?*" We find replication and recursion in MPST to be mutually non-inclusive—*i.e.,* both can produce protocols which *cannot* be typed under the other construct. We have already demonstrated this in one direction with the binary tree examples; below we showcase how recursion cannot be replaced by replication.

Example 10 (Lock Service). The lock service provides clients with a *mutex lock*.

$$\boldsymbol{L} : !\boldsymbol{\theta}\&\mathsf{lock} \,.\, \mu\mathrm{t}.\, \boldsymbol{\theta}\&\,\{\mathsf{acquire} \,.\, \boldsymbol{\theta}\&\mathsf{release} \,.\, \mathrm{t},\ \mathsf{done} \,.\, \mathbf{end}\}$$

When a client requests a lock from $\boldsymbol{L}$, a copy of the recursive continuation is exposed. The recursive definition allows sequences of acquire and release messages to be received. It is key to note that, whilst replication maintains a top-level branch that is permanently available to receive a message, the top-level action in a recursive definition is *not* fixed.

Copies of the continuation type of a replicated receive are executed concurrently. Example 10 provides a service for roles to enter race-sensitive portions of a protocol, as if it were an atomic action. We demonstrate its use by typing the dining philosophers problem.

Example 11 (Dining Philosophers). A number of philosophers gather to eat on a round table. Each plate is separated by a single chopstick, and a philosopher needs two chopsticks to eat. The dining philosophers problem requires the philosophers to employ an algorithm to ensure the table does not get deadlocked. In such a setting, we can view *chopsticks as services* and *philosophers as clients*. Assuming a size of n-philosophers, we define the type for a chopstick as:

$$(C_i)_{1..n} : L\oplus\mathsf{lock}\,.\,!\eta\&\left\{\begin{array}{l}\mathsf{take?}\,.\,L\oplus\mathsf{acquire}\,.\,\eta\oplus\mathsf{ok}\,.\,\eta\&\mathsf{give}\,.\,L\oplus\mathsf{release}\,.\,\mathbf{end}\\ \mathsf{done}\,.\,L\oplus\mathsf{done}\,.\,\mathbf{end}\end{array}\right\}$$

Before offering its service, a chopstick requests a lock from L. This ensures that every chopstick has its own lock that it may acquire and release. The lock is used to guarantee that a chopstick is only ever taken by a single philosopher at a time. A chopstick then waits for a take? request from a philosopher; receiving one will result in it attempting to acquire the lock. This acquisition is only successful if the same role has not already requested it in some other parallel composition. If the lock was already acquired, then the $L\oplus\mathsf{acquire}$ will block until the lock is released. Acquiring the lock sends an ok back to the philosopher, symbolising that they have successfully obtained the chopstick. When done from eating, the philosopher may then send back a give message, which in turn releases the lock, as the chopstick is now available to be taken by a different philosopher.

We can now write an algorithm for philosophers. First, a naive approach:

$$(p_i)_{i\in 1..n} : C_i\oplus\mathsf{take?}.\,C_{i+1}\oplus\mathsf{take?}.\,C_i\&\mathsf{ok}.\,C_{i+1}\&\mathsf{ok}.\,C_i\oplus\mathsf{give}.\,C_{i+1}\oplus\mathsf{give}.\,q\oplus\mathsf{fin}.\,\mathbf{end}$$
$$q : p_1\&\mathsf{fin}.\,\cdots\,.\,p_n\&\mathsf{fin}.\,C_1\oplus\mathsf{done}.\,\cdots\,.\,C_n\oplus\mathsf{done}.\,\mathbf{end}$$

Every philosopher p_i has a similar type. They begin by requesting to take the chopsticks to their left and right—note that this results in every chopstick receiving two take? requests. Receiving both ok messages means the philosopher can eat, and subsequently give back the chopsticks. Finally, when finished, philosophers send a fin to role q, acting as a clean-up for the protocol. The protocol $\{p_i, q, C_i, L\}_{i\in 1..n}$ is safe, but fails typechecking for $\varphi = \mathit{terminating}$. In fact, the naïve protocol allows for scenarios in which all philosophers take a single chopstick, resulting in a deadlock. This problem has many solutions; we present the simplest in which philosophers take turns to eat. (Key changes are underlined.)

$$S_1 = C_1\oplus\mathsf{take?}.\,C_2\oplus\mathsf{take?}.\,C_1\&\mathsf{ok}.\,C_2\&\mathsf{ok}.\,C_1\oplus\mathsf{give}.\,C_2\oplus\mathsf{give}.\,\underline{p_2\oplus\mathsf{fin}}.\,\mathbf{end}$$
$$S_2 = \underline{p_{i-1}\&\mathsf{fin}}.C_i\oplus\mathsf{take?}.C_{i+1}\oplus\mathsf{take?}.C_i\&\mathsf{ok}.C_{i+1}\&\mathsf{ok}.C_i\oplus\mathsf{give}.C_{i+1}\oplus\mathsf{give}.\underline{p_{i+1}\oplus\mathsf{fin}}.\mathbf{end}$$
$$S_3 = \underline{p_{n-1}\&\mathsf{fin}}.C_n\oplus\mathsf{take?}.C_1\oplus\mathsf{take?}.C_n\&\mathsf{ok}.C_1\&\mathsf{ok}.C_n\oplus\mathsf{give}.C_1\oplus\mathsf{give}.q\oplus\mathsf{fin}.\mathbf{end}$$
$$q' : p_n\&\mathsf{fin}.\,C_1\oplus\mathsf{done}.\,\cdots\,.\,C_n\oplus\mathsf{done}.\,\mathbf{end}$$

Here, all philosophers other than the first must wait for the previous to finish eating before they can request to take their chopsticks. The updated protocol $\{p_1 : S_1, p_i : S_2, p_n : S_3, q', C_j, L\}^{i\in 2..n\text{-}1}_{j\in 1..n}$ now typechecks for $\varphi = \mathit{terminating}$.

The previous examples demonstrate how recursion hidden by a universal receive can be used to mimic changes in state. Our final example does the inverse, *i.e.,* we show how a universal receive hidden by a recursive binder can be used to model resources which eventually reach some permanent state. In addition, we show that universal receives model *fair races*, since they do not impose an order on how communication is handled.

Example 12 (Auction). A merchant $\boldsymbol{m}$ sets up an auction $\boldsymbol{A}$ to accept bids from some buyers $\boldsymbol{b}$. A merchant can employ different mechanisms for choosing who to sell to (*e.g.*, first come first served, highest bid, biased selling, *etc.*); but must always respond to buyers with either a yes, no, or not-avail message.

$$\boldsymbol{A} : !\boldsymbol{\alpha}\&\mathsf{bid}(\mathsf{int}).\, \boldsymbol{m}\oplus\mathsf{bid}(\mathsf{int}, \boldsymbol{\alpha}).\, \mathbf{end}$$

$$\boldsymbol{m} : \mu\mathrm{t}.\, \boldsymbol{A}\&\mathsf{bid}(\mathsf{int}, \boldsymbol{\beta}).\, \boldsymbol{\beta}\oplus \left\{ \begin{array}{l} \mathsf{yes}.\, !\boldsymbol{A}\&\mathsf{bid}(\mathsf{int}, \boldsymbol{\kappa}).\, \boldsymbol{\kappa}\oplus\mathsf{not\text{-}avail}.\, \mathbf{end} \\ \mathsf{no}.\, \mathrm{t} \end{array} \right\}$$

$$(\boldsymbol{b_i})_{i\in 1..n} : \mu\mathrm{t}.\, \boldsymbol{A}\oplus\mathsf{bid}(\mathsf{int}).\, \boldsymbol{m}\&\, \{\mathsf{yes}.\, \mathbf{end},\;\; \mathsf{no}.\, \mathrm{t},\;\; \mathsf{not\text{-}avail}.\, \mathbf{end}\}$$

Buyers $\boldsymbol{b_i}$ race to send bids to the auction service. In turn, the auction forwards bids and buyer role identifiers to the merchant, who processes bids sequentially (but still in an arbitrary order). If the merchant declines a bid, then the client is offered another chance; if the merchant accepts a bid, then it exposes a replicated receive which informs any further buyers that the product is no longer available. It is key to note that, unlike in Example 11 where we used locks to avoid race conditions, races here are not only allowed but are integral to the protocol. Additionally, by uncovering a replicated receive, the merchant enters a *permanent* state. These two characteristics guarantee that, no matter the selling algorithm employed by the merchant: *(i)* bids always arrive in a fair arbitrary order; and *(ii)* the product can only be sold once.

Discussion. We have now shown that *replication* and *recursion* are mutually non-inclusive, and that our extension increases the expressiveness of MPST. It is important to understand the dependencies between added features and the expressiveness gained. Since MPST! is a *conservative* extension, it is guaranteed that the increase of expressiveness derives from our two extensions: *1.* the addition of *replication*; and *2.* the addition of *first-class roles*.

Replication alone is enough to increase the expressiveness of MPSTs w.r.t. the Chomsky hierarchy. We note that, *e.g.*, Example 8 could be easily re-written without the universal receive, especially since it should not be used by multiple clients to uphold deadlock-freedom—thus, *replication in MPSTs increases their expressiveness to that of context-free languages.*

First-class roles in our formalism refers to: *(i) universal receives* acting as binders on role variables; and *(ii)* the ability to *pass roles* as payloads in messages. Universal receives allow *protocols to be designed agnostic of the client pool* (consider Examples 2, 6, 9 and 10); and also act as a *fair way of introducing races*—*e.g.* Example 2 describes a load balancer that responds to requests in *no particular order.* Role passing allows for *safe distributed choice.* In a load balancer (in general), it is *impossible* for a client to know which worker will service its request. In Example 2, role passing allows the server to inform clients of its choice, and also informs the worker of the identity of the client. Role passing increases expressiveness by introducing dependencies into a protocol. For instance, Example 12 uses role passing to ensure the merchant correctly services the right buyers; without it, a merchant could not respond to requests without bias.

As a final note, first-class roles are different to, *e.g.*, *delegation* or *multiple sessions* (both supported in MPST!) since they act *inside* a session. Therefore our system can still be used to check for properties such as deadlock-freedom, which is not possible with interleaved sessions without other mechanisms such as an interaction typing system [3] or priorities [8].

3.2 Decidability

As one may expect, the added expressiveness of replication and first-class roles into types does not come without a cost. Unlike the base theory that this work extends, even though our language models synchronous communication, typechecking may be *undecidable* in the general case. In the following, we discuss decidability of typechecking in detail. We show that typechecking is only as decidable as the safety property; we provide examples of types that make typechecking problematic; and we provide strategies for determining whether a protocol is captured by a decidable subset of MPST!.

Theorem 3 (Decidablility of type checking). *If φ is decidable, then typechecking is decidable.*

Proof. Since typing rules in Figure 6 can be deterministically applied based on the structure of a process P, and a typing context need only be split a finite number of times to separate all linear types, there are a finite number of contexts that can be tried for each rule that requires a context split. Lastly, subtyping is decidable [13] (decidability of subtyping replicated types is equivalent to regular branching types, and of parallel types is equivalent to checking multiple session types); and φ is decidable by assumption. ◻

Theorem 3 states that decidability of type checking is only as decidable as property φ. In Example 14, we will demonstrate why φ may not necessarily be decidable in the general case for the type semantics presented in Figure 7. To do this, we first define *behavioural sets* of type contexts (as in [25, appendix K]).

Definition 8 (Behavioural set). *The behavioural set of a type context, written $\mathsf{beh}(\Gamma)$, is given by $\mathsf{beh}(\Gamma) = \mathsf{unf}^*(\{\Gamma' \mid \Gamma \to^* \Gamma'\})$; where unf^* is the closure of unf—a function that unfolds all top-level recursive binders in a set of contexts. (Full definitions of unf and unf^* are standard and given in Appendix D.)*

Informally, the behavioural set of a context Γ is the set of *(i)* its reductions; and *(ii)* its reductions' unfoldings. The benefit of beh is that it mechanically abides by conditions [S-μ] and [S-$\to$] from Definition 6. Therefore, to determine whether $\mathsf{beh}(\Gamma)$ is a safety property, all that is required is to exhaustively check the contexts that inhabit $\mathsf{beh}(\Gamma)$ against the remaining conditions of Definition 6.

Example 13. Consider a context $\Gamma = \{s[\mathbf{p}] : \mu t.\mathbf{q}{\oplus}\mathsf{m}.t, s[\mathbf{q}] : \mu t'.\mathbf{p}\&\mathsf{m}.t'\}$. The behavioural set of Γ is given by:

$$\mathsf{beh}(\Gamma) = \left\{ \left\{ \begin{array}{l} s[\mathbf{p}] : \mu t.\mathbf{q}{\oplus}\mathsf{m}.t, \\ s[\mathbf{q}] : \mu t'.\mathbf{p}\&\mathsf{m}.t' \end{array} \right\}, \left\{ \begin{array}{l} s[\mathbf{p}] : \mathbf{q}{\oplus}\mathsf{m}.\mu t.\mathbf{q}{\oplus}\mathsf{m}.t, \\ s[\mathbf{q}] : \mathbf{p}\&\mathsf{m}.\mu t'.\mathbf{p}\&\mathsf{m}.t' \end{array} \right\} \right\}$$

Notice that the left element is the original context after 0 reduction steps, whereas the right element is the unfolding of Γ. Moreover, any further reductions only yield contexts (and unfoldings) already captured by these two elements.

The next example context is problematic for typechecking.

Example 14. Consider a context $\Gamma = \{s[\mathbf{p}] : \mu t.\mathbf{q}{\oplus}\mathsf{m}.t, s[\mathbf{q}] : !\mathbf{p}\&\mathsf{m}.\mathbf{r}{\oplus}\mathsf{m}\}$. The behavioural set of Γ is given by:

$$\mathsf{beh}(\Gamma) = \left\{ \begin{array}{l} \left\{ \begin{array}{l} s[\mathbf{p}] : \mu t.\mathbf{q}{\oplus}\mathsf{m}.t, \\ s[\mathbf{q}] : !\mathbf{p}\&\mathsf{m}.\mathbf{r}{\oplus}\mathsf{m} \end{array} \right\}, \left\{ \begin{array}{l} s[\mathbf{p}] : \mathbf{q}{\oplus}\mathsf{m}.\mu t.\mathbf{q}{\oplus}\mathsf{m}.t, \\ s[\mathbf{q}] : !\mathbf{p}\&\mathsf{m}.\mathbf{r}{\oplus}\mathsf{m} \end{array} \right\}, \\ \left\{ \begin{array}{l} s[\mathbf{p}] : \mu t.\mathbf{q}{\oplus}\mathsf{m}.t, \\ s[\mathbf{q}] : !\mathbf{p}\&\mathsf{m}.\mathbf{r}{\oplus}\mathsf{m} \mid \mathbf{r}{\oplus}\mathsf{m} \end{array} \right\}, \left\{ \begin{array}{l} s[\mathbf{p}] : \mathbf{q}{\oplus}\mathsf{m}.\mu t.\mathbf{q}{\oplus}\mathsf{m}.t, \\ s[\mathbf{q}] : !\mathbf{p}\&\mathsf{m}.\mathbf{r}{\oplus}\mathsf{m} \mid \mathbf{r}{\oplus}\mathsf{m} \end{array} \right\}, \ldots \end{array} \right\}$$

Indeed, $\mathsf{beh}(\Gamma)$ is *infinite*. This is a result of how replication and parallel composition are modelled in Figure 7. In fact, the type semantics for replicated communication allows for context reduction to yield *larger* types. Note how in this example, the contexts that inhabit $\mathsf{beh}(\Gamma)$ get infinitely larger by pulling out infinitely many copies of type $\mathbf{r}{\oplus}\mathsf{m}$.

Furthermore, we point out that infinite behavioural sets are not only a result of recursive communication with replicated branches. Consider, *e.g.* a $\Gamma' = \{s[\mathbf{p}] : !\boldsymbol{\alpha}\&\mathsf{m}.\boldsymbol{\alpha}{\oplus}\mathsf{m}'.\mathbf{r}{\oplus}\mathsf{m}, s[\mathbf{q}] : \mathbf{p}{\oplus}\mathsf{m}.!\boldsymbol{\beta}\&\mathsf{m}'.\boldsymbol{\beta}{\oplus}\mathsf{m}\}$. Such a context will also pull out infinitely many copies of type $\mathbf{r}{\oplus}\mathsf{m}$, because the replicated communication forms an infinite loop. Lastly, it is key to note that $\mathsf{beh}(\Gamma'')$ is finite for any Γ'' that does not contain replicated branches, since there is no other way for a context reduction to yield a larger type.

Knowing whether $\mathsf{beh}(\Gamma)$ is (in-)finite is key for our main decidability result.

Theorem 4 (Decidability of beh). *Let $\varphi = \mathsf{beh}(\Gamma)$. If $\mathsf{beh}(\Gamma)$ is finite, then φ is decidable.*

Proof. Since $\mathsf{beh}(\Gamma)$ contains all possible reductums and unfoldings of Γ, then conditions [S-$\rightarrow$] and [S-μ] are satisfied immediately. Therefore, to determine whether $\mathsf{beh}(\Gamma)$ is a safety property, we may exhaustively check all inhabitants of $\mathsf{beh}(\Gamma)$ against conditions [S-$\oplus\&$], [S-$!\oplus\&$], [S-$\boldsymbol{\alpha}$], which is decidable since $\mathsf{beh}(\Gamma)$ is finite (by assumption); and since subtyping and frv are decidable. $\square$

Theorem 4 states that φ is *decidable* for any $\varphi = \mathsf{beh}(\Gamma)$ where $\mathsf{beh}(\Gamma)$ is a finite set. In other words, if a protocol can be shown to have a finite behavioural set, then typechecking for that protocol is *decidable*. This could be done manually for each protocol; however, to further increase the practicality of our type system, we present two strategies for restricting protocols into a subset of MPST! with finite behavioural sets.

Decidability Strategies. The strategies we present for restricting protocols to decidable subsets of MPST! all follow a similar blueprint. Essentially, we wish to establish properties on Γ with decidable approximations that imply that $\mathsf{beh}(\Gamma)$ is finite. Then, by Theorems 3 and 4 we obtain decidable typechecking.

The following defines, and gives examples, of each strategy; then, we show these strategies are sound and discuss how they can be approximated.

Definition 9 prevents types like Γ in Example 14 using a naïve approach; put simply, tf captures protocols where all clients of a replicated server are intrinsically non-recursive and non-replicated.

Definition 9 (Trivially finite). *A context Γ is trivially finite, $\mathsf{tf}(\Gamma)$, iff:*

1. *no type in the body of a recursive binder sends to a replicated branch; and*
2. *continuations of replicated branches do not send to other replicated branches.*

Example 15. The protocols modelling the dining philosophers problem in Example 11 are *trivially finite*. Note how the chopstick services make the initial request to the lock service *before* they offer their replicated branch.

For other protocols we need a more nuanced strategy. Definition 10 formalises "loops" in a protocol which may result from replicated servers infinitely bouncing messages amongst each other (such as Γ' in Example 14).

Definition 10 (Loop free). *Given a protocol Ψ, and a context Γ derived from Ψ (possibly after a number of reductions), a* ***cycle*** *in the LTS of Γ is defined as the series of transitions s.t., for $\Gamma = \Gamma' \cdot s[\mathbf{p}] : !\rho \&_{i \in I} \mathsf{m}_i(\widetilde{T_i}).S_i$*

$$\Gamma \xrightarrow{s:\mathbf{q},\mathbf{p}:\mathsf{m}_k} \left(\xrightarrow{s:\mathbf{p}'_j,\mathbf{q}'_j:\mathsf{m}'_j} \right)_{j \in 1..n} \xrightarrow{s:\mathbf{q},\mathbf{p}:\mathsf{m}_k} \Gamma''$$

*where $k \in I$, and for any $\mathbf{p}', \mathbf{q}', \mathsf{m}', n, \Gamma''$. A cyclic replicated communication path (**CRCP**) is defined as a cycle with these added conditions:*

1. *$\Gamma(s[\mathbf{q}]) = S_{\mathbf{q}}$ s.t. either $S_{\mathbf{q}} \equiv !S^{\&} \mid U$ or $S_{\mathbf{q}}$ appears after a recursive binder in $\Psi(\mathbf{q})$, for any $S^{\&}, U$; and*
2. *$\forall x \in 1..n : \Gamma \xrightarrow{s:\mathbf{q},\mathbf{p}:\mathsf{m}_k} \left(\xrightarrow{s:\mathbf{p}'_l,\mathbf{q}'_l:\mathsf{m}'_l} \right)_{l \in 1..x-1} \Gamma''' \xrightarrow{s:\mathbf{p}'_x,\mathbf{q}'_x:\mathsf{m}'_x}$ we have: $\Gamma'''(s[\mathbf{q}'_x]) = S_{\mathbf{q}'_x}$ s.t. either $S_{\mathbf{q}'_x} \equiv !S^{\&} \mid U$ or $S_{\mathbf{q}'_x}$ appears after a recursive binder in $\Psi(\mathbf{q}'_x)$.*

*We say Γ is **loop free**, written $\mathsf{lf}(\Gamma)$, iff the LTS of Γ does not contain a CRCP.*

Essentially, a *cycle* in the LTS of a context Γ: *(i)* starts with an incoming communication action into a replicated type; *(ii)* performs some intermediary transitions; and *(iii)* ends with the transition that began the cycle. A CRCP is a special case of a cycle, where all intermediary transitions must also be between roles that have a replicated type; or form part of the body of a recursive type. Finally, a context is *loop free* iff its LTS does not produce any CRCPs.

Example 16. Contexts Γ and Γ' from Example 14 contain CRCPs: Γ contains a CRCP at $\boldsymbol{q}$ with 0 intermediary transitions; and Γ' contains two CRCPs, at $\boldsymbol{q}$ and $\boldsymbol{p}$, both with 1 intermediary transition forming part of a replicated type.

Example 17. The protocols in Examples 2, 8 and 9 are *loop free*: Example 2 because there are no cycles; Examples 8 and 9 because the cycle between the pong branch on $\boldsymbol{T}$ (resp. $\boldsymbol{M}$) and the ping branch on $\boldsymbol{P}$ (resp. $\boldsymbol{p_1}, \boldsymbol{p_2}$) includes communication with the (non-replicated/-recursive) client; breaking the CRCP.

Proposition 1. *If* $\mathsf{beh}(\Gamma)$ *is infinite, then* Γ *contains a CRCP.*

Proof. From the type semantics (Figure 7), we observe that the only reductions that can yield *larger* types are communications with replicated branches. Therefore, it follows that for $\mathsf{beh}(\Gamma)$ to be infinite, there must be some reoccurring transitions in the LTS of Γ that repeatedly communicates with a replicated branch—*i.e.,* there is a communication action on a replicated branch, followed by any number of intermediary transitions which then end with the initial communication action on the replicated branch; where all intermediary transitions must be non-finite. This is the definition of a CRCP (Definition 10). ◻

Theorem 5 (Strategy soundness). *Given a context* Γ*,* $\Phi(\Gamma)$ *implies* $\mathsf{beh}(\Gamma)$ *is finite, for* $\Phi \in \{\mathsf{tf}, \mathsf{lf}\}$

Proof. Case tf. By contradiction: assume $\mathsf{beh}(\Gamma)$ is infinite, then, by Proposition 1, Γ contains a CRCP; but, by Definition 10, a CRCP will violate at least one of the conditions for tf in Definition 9—contradiction.
Case lf. By contradiction: assume $\mathsf{beh}(\Gamma)$ is infinite, then, by Proposition 1, Γ contains a CRCP—contradiction; therefore $\mathsf{beh}(\Gamma)$ is *finite.* ◻

Approximations. Properties tf and lf are both *decidable* for all protocols *without* first-class roles: tf can be determined via a linear traversal of a type context; and lf can be checked by constructing a directed graph of visited replicated branches in a context, then checking that the graph is acyclic (which is decidable). An approximation is only required for protocols using role variables, since their values can only be known at runtime. This approximation would treat any role variable in a selection type to have the capability of reaching *any other role*.

Example 18 (Approximation false negative). Consider the following protocol:

$$\boldsymbol{p} : !\boldsymbol{\alpha}\&\mathsf{m}.\boldsymbol{\alpha}\oplus\mathsf{m}' \qquad \boldsymbol{q} : \boldsymbol{p}\oplus\mathsf{m}.\boldsymbol{p}\&\mathsf{m}'.\boldsymbol{r}\oplus\mathsf{m}'.\boldsymbol{r}\&\mathsf{m} \qquad \boldsymbol{r} : !\boldsymbol{\beta}\&\mathsf{m}'.\boldsymbol{\beta}\oplus\mathsf{m}$$

Although the above is *trivially finite* ($\boldsymbol{p}$ and $\boldsymbol{r}$ do not communicate), it would be *falsely* flagged as *not* tf because $\boldsymbol{\alpha}$ is over-approximated to include $\boldsymbol{r}$.

It is key to note that false negatives of the approximation are avoidable by *requiring unique branching labels on replicated types.* Furthermore, even with the approximation, all examples presented in this paper (except Example 12) are captured by either lf or tf. With respect to Example 12, the presented protocol still yields a finite behavioural set, and thus by Theorem 4 and Theorem 3, typechecking it is decidable. We aim to continue exploring further strategies (especially ones which can capture protocols such as Example 12) in future work.

4 Related Work

The MAGπ calculus [19] makes use of generalised MPST theory in order to type *failure-prone* communications (i.e., message loss, reordering, and delays). Key to their approach is the use of timeouts to detect and handle message loss; as with related approaches (e.g., Barwell et al. [2]), this often means that session types are made more complex in order to handle each potential failure point. Our approach is most closely related to that of Le Brun & Dardha [4], who introduce MAGπ! as a modification of MAGπ to incorporate type-level replication: this has the advantage of simplifying client-server interactions by only requiring *clients* to handle potential failures. However, the aims of our work and that of Le Brun & Dardha [4] are significantly different: while their aim is specifically to use replication as a methodology to simplify failure handling, our work is a more fundamental study of the consequences of type-level replication on expressiveness and decidability. In particular we make use of a more standard base MPST calculus (i.e., a calculus that does not include undirected receives, nor rules that model failures and message reordering), and we make use of *synchronous* communication semantics. Nevertheless our calculus allows for more interesting use of replication: unlike MAGπ! we allow *nested replication* and *recursion*, whereas MAGπ! only allows replication at the top-level and processes must be finite. Furthermore, as a result of our inclusion of *first-class roles*, as well as using replication *in tandem with* recursion, we can type protocols that make non-trivial use of mutual exclusion and races, all of which would be *inexpressible in MAGπ!*.

Toninho & Yoshida [29] assess the relative expressiveness of a multiparty session calculus and a process calculus inspired by classical linear logic, showing that MPST calculi allow strictly more expressive process networks (i.e., those that can include cycles). As part of this investigation they explore a limited form of type-level replication that permits a liveness property. However, their system does not consider first-class roles and pre-dates generalised MPST so is guided primarily by global types, and is therefore less expressive than MPST!.

Replicated session types have been used to a limited extent in a wide variety of works on binary session types (e.g., [6,8,5,31]), primarily in works pertaining to Curry-Howard interpretations of propositions as session types, where the exponential modality from linear logic $!A$ is typically linked to replication from the π-calculus. Several further lines of work investigate client-server communication following this correspondence. Kokke et al. [18] investigate an extension of the logically-inspired HCP calculus [17] with two dual modalities $!_n A$ and $?_n A$ to type a pool of n clients and a replicated server that can service n requests respectively, and show that their calculus allows nondeterministic behaviour while still preventing deadlocks and ensuring termination. Qian et al. [22] develop CSLL (client-server linear logic) that uses the dual *coexponential* modalities $¡A$ and $¿A$ to type servers and client pools respectively, along with rules to merge client pools. The subtle difference between the $¡A$ modality and the exponential $!A$ being that the former (informally) serves type A *only as many times as required according to client requests*. This is similar to how our type system operates, given that replicated receives only pull out copies of continuations upon communication.

Multiple requests induce non-determinism into further reductions; in our work this is observed through parallel types, which in the work of Qian et al. [22] is observed through hyperenvironments [17,12]. Unlike all these works, we focus on *multiparty* session types, where interacting with a replicated channel spawns a process that *remains in the same session*. This results in our key novelty, *i.e.,* our account of replication in the *type semantics* with the use of parallel types.

Marshall & Orchard [20] investigate the effects of adding a *semiring graded necessity* modality (a generalisation of the ! modality) to a session-typed λ-calculus, showing interesting consequences such as replicated servers and multicast communication. We posit that the two systems can type different protocols: while MPST! cannot straightforwardly encode multicast communication, it is difficult to see how their approach would scale to the examples we describe in Section 3.

Rocha & Caires [23] introduce CLASS, a process calculus with a correspondence to Differential Linear Logic [11]. CLASS integrates session-typed communication, reference cells with mutual exclusion, and replication. Their calculus guarantees preservation and progress, the proof of the latter property requiring a logical relation. CLASS can encode the dining philosophers problem, making essential use of shared state; in contrast our implementation relies on the interplay between replication and recursion.

Deniélou et al. [10] introduce parameterised MPST as a means of designing protocols for parallel algorithms. Their formalism allows for parameterisation of participants in the form of $\mathit{client}[i]$, representing the i^{th} client from some group of n clients, for a bound n. The key difference between this formalism and MPST! is that our approach preserves, and allows for the fair handling of, *races*. Parameterised MPST enforce a predetermined prioritisation on the order of communication (thus, Example 12 cannot be expressed in that system).

5 Conclusion

We presented MPST!, a conservative extension of the standard multiparty session π-calculus which introduces for the first time *replication* and *first-class roles*, and proved its metatheory. We have shown that the interplay between replication and recursion allows us to describe interesting and previously inexpressible MPST protocols such as those that rely on races and mutual exclusion, as well as giving a method by which we can express context-free protocols. Although replication can have implications for decidability of typechecking, we have identified sufficient conditions that can determine decidability and provided syntactic approximations for decidability. For future work, it would be interesting to investigate an extension of MPST! with polymorphism, as this would improve on the modular design of protocols already promoted by the type system. Furthermore, we wish to continue exploring the decidability of typechecking to find more general approximations.

Acknowledgements. We thank the anonymous reviewers for their detailed and insightful comments. This work was supported by EPSRC Grants EP/X027309/1 (Uni-pi) and EP/T014628/1 (STARDUST).

References

1. Almeida, B., Mordido, A., Thiemann, P., Vasconcelos, V.T.: Polymorphic lambda calculus with context-free session types. Inf. Comput. **289**(Part), 104948 (2022). https://doi.org/10.1016/j.ic.2022.104948, https://doi.org/10.1016/j.ic.2022.104948
2. Barwell, A.D., Scalas, A., Yoshida, N., Zhou, F.: Generalised multiparty session types with crash-stop failures. In: Klin, B., Lasota, S., Muscholl, A. (eds.) 33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland. LIPIcs, vol. 243, pp. 35:1–35:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.CONCUR.2022.35, https://doi.org/10.4230/LIPIcs.CONCUR.2022.35
3. Bettini, L., Coppo, M., D'Antoni, L., Luca, M.D., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5201, pp. 418–433. Springer (2008). https://doi.org/10.1007/978-3-540-85361-9_33, https://doi.org/10.1007/978-3-540-85361-9_33
4. Brun, M.A.L., Dardha, O.: Magπ!: The role of replication in typing failure-prone communication. In: Castiglioni, V., Francalanza, A. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 44th IFIP WG 6.1 International Conference, FORTE 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings. Lecture Notes in Computer Science, vol. 14678, pp. 99–117. Springer (2024). https://doi.org/10.1007/978-3-031-62645-6_6, https://doi.org/10.1007/978-3-031-62645-6_6
5. Caires, L., Pérez, J.A.: Linearity, control effects, and behavioral types. In: Yang, H. (ed.) Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10201, pp. 229–259. Springer (2017). https://doi.org/10.1007/978-3-662-54434-1_9, https://doi.org/10.1007/978-3-662-54434-1_9
6. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Math. Struct. Comput. Sci. **26**(3), 367–423 (2016). https://doi.org/10.1017/S0960129514000218, https://doi.org/10.1017/S0960129514000218
7. Coppo, M., Dezani-Ciancaglini, M., Padovani, L., Yoshida, N.: A gentle introduction to multiparty asynchronous session types. In: Bernardo, M., Johnsen, E.B. (eds.) Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures. Lecture Notes in Computer Science, vol. 9104, pp. 146–178. Springer (2015). https://doi.org/10.1007/978-3-319-18941-3_4, https://doi.org/10.1007/978-3-319-18941-3_4
8. Dardha, O., Gay, S.J.: A new linear logic for deadlock-free session-typed processes. In: Baier, C., Lago, U.D. (eds.) Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10803, pp. 91–109. Springer (2018). https://doi.org/10.1007/978-3-319-89366-2_5, https://doi.org/10.1007/978-3-319-89366-2_5

9. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. Inf. Comput. **256**, 253–286 (2017). https://doi.org/10.1016/j.ic.2017.06.002, https://doi.org/10.1016/j.ic.2017.06.002
10. Deniélou, P., Yoshida, N., Bejleri, A., Hu, R.: Parameterised multiparty session types. Log. Methods Comput. Sci. **8**(4) (2012). https://doi.org/10.2168/LMCS-8(4:6)2012, https://doi.org/10.2168/LMCS-8(4:6)2012
11. Ehrhard, T.: An introduction to differential linear logic: proof-nets, models and antiderivatives. Math. Struct. Comput. Sci. **28**(7), 995–1060 (2018)
12. Fowler, S., Kokke, W., Dardha, O., Lindley, S., Morris, J.G.: Separating sessions smoothly. Log. Methods Comput. Sci. **19**(3) (2023)
13. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta Informatica **42**(2-3), 191–225 (2005). https://doi.org/10.1007/S00236-005-0177-Z, https://doi.org/10.1007/s00236-005-0177-z
14. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings. Lecture Notes in Computer Science, vol. 715, pp. 509–523. Springer (1993). https://doi.org/10.1007/3-540-57208-2_35, https://doi.org/10.1007/3-540-57208-2_35
15. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1381, pp. 122–138. Springer (1998). https://doi.org/10.1007/BFb0053567, https://doi.org/10.1007/BFb0053567
16. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016). https://doi.org/10.1145/2827695, https://doi.org/10.1145/2827695
17. Kokke, W., Montesi, F., Peressotti, M.: Better late than never: a fully-abstract semantics for classical processes. Proc. ACM Program. Lang. **3**(POPL), 24:1–24:29 (2019)
18. Kokke, W., Morris, J.G., Wadler, P.: Towards races in linear logic. Log. Methods Comput. Sci. **16**(4) (2020)
19. Le Brun, M.A., Dardha, O.: Magπ: Types for failure-prone communication. In: Wies, T. (ed.) Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13990, pp. 363–391. Springer (2023). https://doi.org/10.1007/978-3-031-30044-8_14, https://doi.org/10.1007/978-3-031-30044-8_14
20. Marshall, D., Orchard, D.: Replicate, reuse, repeat: Capturing non-linear communication via session types and graded modal types. In: Carbone, M., Neykova, R. (eds.) Proceedings of the 13th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES@ETAPS 2022, Munich, Germany, 3rd April 2022. EPTCS, vol. 356, pp. 1–11 (2022). https://doi.org/10.4204/EPTCS.356.1, https://doi.org/10.4204/EPTCS.356.1
21. Poças, D., Costa, D., Mordido, A., Vasconcelos, V.T.: System $\mathrm{f}^{\mu}_{\omega}$ with context-free session types. In: Wies, T. (ed.) Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France,

April 22-27, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13990, pp. 392–420. Springer (2023). https://doi.org/10.1007/978-3-031-30044-8_15, https://doi.org/10.1007/978-3-031-30044-8_15
22. Qian, Z., Kavvos, G.A., Birkedal, L.: Client-server sessions in linear logic. Proc. ACM Program. Lang. **5**(ICFP), 1–31 (2021). https://doi.org/10.1145/3473567, https://doi.org/10.1145/3473567
23. Rocha, P., Caires, L.: Safe session-based concurrency with shared linear state. In: ESOP. Lecture Notes in Computer Science, vol. 13990, pp. 421–450. Springer (2023)
24. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming (artifact). Dagstuhl Artifacts Ser. **3**(2), 03:1–03:2 (2017). https://doi.org/10.4230/DARTS.3.2.3, https://doi.org/10.4230/DARTS.3.2.3
25. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. Tech. Rep. 6, Imperial College London (2018), https://www.doc.ic.ac.uk/research/technicalreports/2018/6
26. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. Proc. ACM Program. Lang. **3**(POPL), 30:1–30:29 (2019). https://doi.org/10.1145/3290343, https://doi.org/10.1145/3290343
27. Stutz, F.: Implementability of Asynchronous Communication Protocols - The Power of Choice. Ph.D. thesis, Kaiserslautern University of Technology, Germany (2024), https://kluedo.ub.rptu.de/frontdoor/index/index/docId/8077
28. Thiemann, P., Vasconcelos, V.T.: Context-free session types. In: Garrigue, J., Keller, G., Sumii, E. (eds.) Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016. pp. 462–475. ACM (2016). https://doi.org/10.1145/2951913.2951926, https://doi.org/10.1145/2951913.2951926
29. Toninho, B., Yoshida, N.: Interconnectability of session-based logical processes. ACM Trans. Program. Lang. Syst. **40**(4), 17:1–17:42 (2018)
30. Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. **217**, 52–70 (2012). https://doi.org/10.1016/J.IC.2012.05.002, https://doi.org/10.1016/j.ic.2012.05.002
31. Wadler, P.: Propositions as sessions. J. Funct. Program. **24**(2-3), 384–418 (2014). https://doi.org/10.1017/S095679681400001X, https://doi.org/10.1017/S095679681400001X

A Structural Congruence

Definition 11 (Structural Congruence).

$$P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad P \mid \mathbf{0} \equiv P \qquad (\nu s)\, \mathbf{0} \equiv \mathbf{0} \qquad (\nu s)(\nu s')\, P \equiv (\nu s')(\nu s)\, P$$

$$\frac{s \notin \mathsf{fs}(P)}{(\nu s)\,(P \mid Q) \equiv P \mid (\nu s)\, Q} \qquad \mathit{def}\ \mathrm{X}(\widetilde{x}) = P\ \mathit{in}\ \mathbf{0} \equiv \mathbf{0} \qquad \frac{s \notin \mathsf{fs}(P)}{\mathit{def}\ \mathrm{X}(\widetilde{x}) = P\ \mathit{in}\ (\nu s)\, Q \equiv (\nu s)\,(\mathit{def}\ \mathrm{X}(\widetilde{x}) = P\ \mathit{in}\ Q)}$$

$$\frac{\mathsf{dpv}(D) \cap \mathsf{fpv}(Q) = \emptyset}{\mathit{def}\ D\ \mathit{in}\ (P \mid Q) \equiv (\mathit{def}\ D\ \mathit{in}\ P) \mid Q} \qquad \frac{(\mathsf{dpv}(D) \cup \mathsf{fpv}(D)) \cap \mathsf{dpv}(D') = (\mathsf{dpv}(D') \cup \mathsf{fpv}(D')) \cap \mathsf{dpv}(D) = \emptyset}{\mathit{def}\ D\ \mathit{in}\ (\mathit{def}\ D'\ \mathit{in}\ P) \equiv \mathit{def}\ D'\ \mathit{in}\ (\mathit{def}\ D\ \mathit{in}\ P)}$$

B Lemmata

B.1 Context Operations

Lemma 1. *$\Gamma = \Gamma_1 \cdot \Gamma_2$ implies $\Gamma_1 + \Gamma_2 = \Gamma$.*

Proof. Holds by induction on the derivation of $\Gamma = \Gamma_1 \cdot \Gamma_2$. Notably, in the cases where $\Gamma = \Gamma', c{:}U$, we observe that context composition ‘,’ implies that $c \notin \mathsf{dom}(\Gamma')$. Thus, we may directly apply the context addition rules to obtain the original context. ☐

B.2 Substitution

Lemma 2. *If $\Gamma = \Gamma_1 \cdot \Gamma_2$, then $\Gamma\{\boldsymbol{q}/\boldsymbol{\alpha}\} = \Gamma_1\{\boldsymbol{q}/\boldsymbol{\alpha}\} \cdot \Gamma_2\{\boldsymbol{q}/\boldsymbol{\alpha}\}$.*

Proof. By induction on the derivation of $\Gamma = \Gamma_1 \cdot \Gamma_2$.

Case $\emptyset$: $\emptyset = \emptyset \cdot \emptyset \implies \emptyset\{\boldsymbol{q}/\boldsymbol{\alpha}\} = \emptyset\{\boldsymbol{q}/\boldsymbol{\alpha}\} \cdot \emptyset\{\boldsymbol{q}/\boldsymbol{\alpha}\}$.

Case cL: $\Gamma = \Gamma', c : U$ and $\dfrac{\Gamma' = \Gamma_1 \cdot \Gamma_2}{\Gamma', c{:}U = \Gamma_1, c{:}U \cdot \Gamma_2}$. We observe that:

$$(\Gamma', c{:}U)\{\boldsymbol{q}/\boldsymbol{\alpha}\} = \Gamma'\{\boldsymbol{q}/\boldsymbol{\alpha}\}, c{:}U\{\boldsymbol{q}/\boldsymbol{\alpha}\} = \Gamma'\{\boldsymbol{q}/\boldsymbol{\alpha}\} \cdot c{:}U\{\boldsymbol{q}/\boldsymbol{\alpha}\} \quad \text{(by } \mathsf{frv} \text{ and def of } \cdot\text{)} \tag{4}$$

$$\Gamma'\{\boldsymbol{q}/\boldsymbol{\alpha}\} = \Gamma_1\{\boldsymbol{q}/\boldsymbol{\alpha}\} \cdot \Gamma_2\{\boldsymbol{q}/\boldsymbol{\alpha}\} \quad \text{(by the ind. hyp.)} \tag{5}$$

$$c \notin \mathsf{dom}(\Gamma'),\ \therefore c \notin \mathsf{dom}(\Gamma_1) \quad \text{(from hyp.)} \tag{6}$$

$$(\Gamma', c{:}U)\{\boldsymbol{q}/\boldsymbol{\alpha}\} = \Gamma_1\{\boldsymbol{q}/\boldsymbol{\alpha}\} \cdot \Gamma_2\{\boldsymbol{q}/\boldsymbol{\alpha}\} \cdot c{:}U\{\boldsymbol{q}/\boldsymbol{\alpha}\} \quad \text{(by (4),(5))} \tag{7}$$

$$= \Gamma_1\{\boldsymbol{q}/\boldsymbol{\alpha}\} \cdot c{:}U\{\boldsymbol{q}/\boldsymbol{\alpha}\} \cdot \Gamma_2\{\boldsymbol{q}/\boldsymbol{\alpha}\} \quad \text{(since } \cdot \text{ commutative)} \tag{8}$$

$$= \Gamma_1\{\boldsymbol{q}/\boldsymbol{\alpha}\}, c{:}U\{\boldsymbol{q}/\boldsymbol{\alpha}\} \cdot \Gamma_2\{\boldsymbol{q}/\boldsymbol{\alpha}\} \quad \text{(by (6))} \tag{9}$$

Cases cR and $\mid$ are similar to case cL.

Case $\boldsymbol{\alpha}$: Follows directly from the ind. hyp. since role substitution does not affect singletons. ☐

Lemma 3. *If $\mathsf{end}(\Gamma)$, then $\Gamma\{\boldsymbol{q}/\boldsymbol{\alpha}\} = \Gamma$.*

Proof. By rule induction on $\mathsf{end}(\Gamma)$, observing that for any context which is end-typed, there are no elements that can be substituted. ☐

Lemma 4. *Assume* $\Gamma \vdash V : T$. *Then,* $\Gamma\{q/\alpha\} \vdash V\{q/\alpha\} : T\{q/\alpha\}$.

Proof. The lemma holds trivially if α does not occur free in T. If α is free in T, then there are two cases to consider. First, when $T = \alpha$.
From the assumption and rule [T-α] we obtain $\alpha : \alpha \vdash \alpha : \alpha$. We must show that $\alpha : \alpha \vdash q : q$, which holds by rules [T-Wkn] and [T-q].
The second case is when $T = S$ for some S in which α occurs free. From assumption and inversion of rule [T-Sub], we know that $\Gamma = c : S'$, $V = c$, and $S' \leqslant S$. Hence, we must prove that $(c : S')\{q/\alpha\} \vdash c : S\{q/\alpha\}$. By the definition of frv, this is the same as proving $c : (S'\{q/\alpha\}) \vdash c : S\{q/\alpha\}$. In turn, by rule [T-Sub], we must prove that $S' \leqslant S$ implies $S'\{q/\alpha\} \leqslant S\{q/\alpha\}$, which holds by lemma 7. □

Lemma 5. *Assume* $\Theta ; \Gamma \vdash P$. *Then,* $\Theta ; \Gamma\{q/\alpha\} \vdash P\{q/\alpha\}$.

Proof. By induction on the derivation of $\Theta ; \Gamma \vdash P$.

Case [T-**0**]: Follows from lemma 3.

Case [T-|]: Follows from the ind. hyp. and lemma 2.

Case [T-+] follows directly from the ind. hyp..

Case [T-ν]: R.T.P. $\Theta ; \Gamma' \vdash (\nu s : \Psi)\ P'$ implies $\Theta ; \Gamma'\{q/\alpha\} \vdash (\nu s : \Psi)\ P'\{q/\alpha\}$. We know:

$$\dfrac{s \notin \Gamma' \qquad \begin{array}{c}\varphi(\mathsf{assoc}_s(\Psi)) \\ \varphi \text{ is a } \textit{safety} \text{ property} \\ \Theta ; \Gamma' + \mathsf{assoc}_s(\Psi) \vdash P'\end{array}}{\Theta ; \Gamma' \vdash (\nu s : \Psi)\ P'} \quad \text{(from assumption)} \quad (10)$$

$$\Theta ; (\Gamma' + \mathsf{assoc}_s(\Psi))\{q/\alpha\} \vdash P'\{q/\alpha\} \quad \text{(ind. hyp.)} \quad (11)$$

But since φ is a *safety* property and $\varphi(\mathsf{assoc}_s(\Psi))$, then by condition [S-$\alpha$], we know that there are *no free role variables in* $\mathsf{assoc}_s(\Psi)$. Hence we infer the below:

$$\Theta ; \Gamma'\{q/\alpha\} + \mathsf{assoc}_s(\Psi) \vdash P'\{q/\alpha\} \quad \text{(by (10), (11) and [S-}\alpha\text{])} \quad (12)$$

$$\Theta ; \Gamma'\{q/\alpha\} \vdash (\nu s : \Psi)\ P'\{q/\alpha\} \quad \text{(by (10), (12) and [T-}\nu\text{])} \quad (13)$$

Case [T-Def]: This case follow directly from the ind. hyp., but we make a note that this is only the case because of how we define role substitution on process definition, *i.e.,*

$$(\mathsf{def}\ \mathrm{X}(\widetilde{x : S}) = P \ \mathsf{in}\ Q)\{q/\alpha\} := \mathsf{def}\ \mathrm{X}(x : \widetilde{S\{q/\alpha\}}) = P\{q/\alpha\} \ \mathsf{in}\ Q\{q/\alpha\}$$

Cases for rules T-!, T-&, T-$\oplus$ and T-Call are similar and all follow from some application of lemma 4. We expand on case [T-$\oplus$] below:

Let $\Gamma = \Gamma' \cdot \Gamma_\oplus \cdot \Gamma_r(\cdot \Gamma_i)_{i \in 1..n}$.
R.T.P. $\Theta ; \Gamma \cdot \Gamma_\oplus \cdot \Gamma_r(\cdot \Gamma_i)_{i \in 1..n} \vdash c[\rho] \oplus \mathsf{m}\langle (V_i)_{i \in 1..n} \rangle . P'$ implies
$\Theta ; (\Gamma' \cdot \Gamma_\oplus \cdot \Gamma_r(\cdot \Gamma_i)_{i \in 1..n})\{q/\alpha\} \vdash (c[\rho] \oplus \mathsf{m}\langle (V_i)_{i \in 1..n} \rangle . P')\{q/\alpha\}$.

We know:

$$\frac{\Gamma_{\oplus} \vdash c : \rho{\oplus}\mathsf{m}(\widetilde{T}).S' \qquad \Gamma_r \vdash \rho : \rho \qquad (\Gamma_i \vdash V_i : T_i)_{i\in 1..n} \qquad \Theta\,;\Gamma' + c : S' \vdash P'}{\Theta\,;\Gamma' \cdot \Gamma_{\oplus} \cdot \Gamma_r (\cdot\, \Gamma_i)_{i\in 1..n} \vdash c[\rho] \oplus \mathsf{m}\langle (V_i)_{i\in 1..n}\rangle\,.\,P'} \quad \text{(from assumption)} \quad (14)$$

$\Gamma_{\oplus}\{q/\alpha\} \vdash c : \rho{\oplus}\mathsf{m}(\widetilde{T}).S'\{q/\alpha\}$ (by (14) and lemma 4) (15)

$\Gamma_r\{q/\alpha\} \vdash (\rho : \rho)\{q/\alpha\}$ (by (14) and lemma 4) (16)

$\forall i \in 1..n : \Gamma_i\{q/\alpha\} \vdash V_i\{q/\alpha\} : T_i\{q/\alpha\}$ (by (14) and lemma 4) (17)

$\Theta\,;(\Gamma' + c : S')\{q/\alpha\} \vdash P'\{q/\alpha\}$ (by (14) and ind. hyp.) (18)

∴ from lines (15)–(18) and rule [T-$\oplus$], we obtain the thesis. ◻

Lemma 6. *Assume $\Theta\,;\Gamma \cdot x : S \vdash P$ and $\Gamma' \vdash s[p] : S''$ where $S'' \leqslant S$. Then, $\Theta\,;\Gamma + \Gamma' \vdash P\{s[p]/x\}$.*

Proof. Similar to [25, Lemma B.1]; in our case we make use of the context operations instead of directly using context compositionssss.

B.3 Subtyping

Lemma 7. *$S \leqslant S'$ implies $S\{q/\alpha\} \leqslant S'\{q/\alpha\}$.*

Proof. By coinduction on the derivation of $S \leqslant S'$. We expand on the branching case below; other cases follow similar reasoning.

Consider the case where $S = \rho\&_{i\in I}\mathsf{m}_i(\widetilde{T_i}).S_i$ and $S' = \rho\&_{i\in I\cup J}\mathsf{m}_i(\widetilde{T_i'}).S_i'$. We know:

$$\frac{(\widetilde{T_i} \leqslant \widetilde{T_i'})_{i\in I} \qquad (S_i'' \leqslant S_i''')_{i\in I}}{\rho\&_{i\in I}\mathsf{m}_i(\widetilde{T_i}).S_i'' \leqslant \rho\&_{i\in I\cup J}\mathsf{m}_i(\widetilde{T_i'}).S_i'''} \quad \text{(by assumption)} \quad (19)$$

If $\rho = \alpha$, then the substitution takes place on both sides, and thus subtyping is preserved top-level. By the coinductive hyp. we know that the property holds for $(S_i'' \leqslant S_i''')_{i\in I}$, and for all payload types that are session types. For any payload types that are *not* session types, we know by definition of subtyping (definition 2) that $\widetilde{T_i} = \widetilde{T_i'}$. Then, if $\widetilde{T_i} = \alpha$, the substitution takes place on both sides, preserving the subtype relation. ◻

Lemma 8. *$\mathsf{end}(\Gamma)$ implies if $\Gamma' \leqslant \Gamma$, then $\Gamma' = \Gamma$.*

Proof. By induction on $\mathsf{end}(\Gamma)$, follows immediately from definition 5.

Lemma 9. *If $\Gamma = \Gamma_1 \cdot \Gamma_2$ and $\Gamma' \leqslant \Gamma$, then $\Gamma' = \Gamma_3 \cdot \Gamma_4$ where $\Gamma_3 \leqslant \Gamma_1$ and $\Gamma_4 \leqslant \Gamma_2$.*

Proof. By induction on $\Gamma = \Gamma_1 \cdot \Gamma_2$, observing that subtyping does not affect the domain of a context, hence subtyping over splits can be preserved.

Lemma 10. *If $\Gamma_1 + \Gamma_2 = \Gamma$ and $\Gamma' \leqslant \Gamma$, then $\exists \Gamma_3$ and Γ_4 s.t. $\Gamma_3 + \Gamma_4 = \Gamma'$ and $\Gamma_3 \leqslant \Gamma_1$ and $\Gamma_4 \leqslant \Gamma_2$.*

Proof. Similar to lemma 9, this time by induction on $\Gamma_1 + \Gamma_2 = \Gamma$.

Lemma 11. *Assume $\Gamma \vdash V : T$ and $\Gamma' \leqslant \Gamma$. Then, $\Gamma' \vdash V : T$.*

Proof. By induction on the typing derivation.

Case [T-q]: $\emptyset \vdash q : q$ and only $\emptyset \leqslant \emptyset$.

Case [T-α]: $\alpha : \alpha \vdash \alpha : \alpha$ but only $\alpha : \alpha \leqslant \alpha : \alpha$.

Case [T-Sub]: $\dfrac{S \leqslant S'}{c : S \vdash c : S'}$. Consider $c : S'' \leqslant c : S$, then by transitivity of subtyping, $\dfrac{S'' \leqslant S'}{c : S'' \vdash c : S'}$.

Case [T-Wkn]: $\dfrac{\Gamma_1 + \Gamma_2 = \Gamma' \quad \Gamma_1 \vdash V : T \qquad \mathsf{end}(\Gamma_2)}{\Gamma' \vdash V : T}$. Consider $\Gamma'' \leqslant \Gamma'$, then by lemma 10, ${\Gamma_1}' + {\Gamma_2}' = \Gamma''$ and ${\Gamma_1}' \leqslant \Gamma_1$ and ${\Gamma_2}' \leqslant \Gamma_2$. By the ind. hyp. we know ${\Gamma_1}' \vdash V : T$; and by lemma 8 we know $\mathsf{end}({\Gamma_2}')$.

Therefore, we conclude by $\dfrac{{\Gamma_1}' + {\Gamma_2}' = \Gamma'' \quad {\Gamma_1}' \vdash V : T \qquad \mathsf{end}({\Gamma_2}')}{\Gamma'' \vdash V : T}$ T-Wkn ◻

Lemma 12. *Assume $\Theta ; \Gamma \vdash P$ and $\Gamma' \leqslant \Gamma$. Then, $\Theta ; \Gamma' \vdash P$.*

Proof. By induction on the typing derivation. Follows from lemma 11. ◻

B.4 Congruence

Lemma 13. *Assume $\Theta ; \Gamma \vdash P$ and $P \equiv P'$. Then, $\exists \Gamma'$ s.t. $\Gamma \equiv \Gamma'$ and $\Theta ; \Gamma' \vdash P'$.*

Proof (Sketch). By cases on the definition of $P \equiv P'$, observing that either a matching congruence can be applied to the type context to preserve the judgement, or the process remains typable under the original context.

B.5 Safety

Lemma 14. *If Γ, Γ' is safe, then Γ is safe.*

Proof. By contradiction. Assume Γ not safe. Then, by definition 6 condition [S-$\rightarrow$], there is a Γ'' s.t. $\Gamma \rightarrow^* \Gamma''$ and Γ'' either violates [S-$\oplus\&$] or [S-α] (possibly after some applications of [S-μ]). But by [Γ-Cong], $\Gamma, \Gamma' = \Gamma \cdot \Gamma' \rightarrow^* \Gamma'' + \Gamma'$ that is not safe. Therefore, Γ, Γ' violates [S-$\rightarrow$] and is itself not safe—contradiction. Hence, Γ, Γ' is safe. ◻

Lemma 15. *If Γ is safe and $\Gamma = s[p] : S_\oplus, s[q] : S_\& \leqslant \Gamma'$ and $\Gamma' \rightarrow$, then $\Gamma \rightarrow$.*

Proof (Sketch). We observe that by virtue of subtyping (definition 2), at least one overlapping message that allows for reduction in the supertype will remain present in the subtype.

Lemma 16. *Assume Γ is safe, $\Gamma \leqslant \Gamma'$, and $\Gamma' \rightarrow \Gamma''$. Then, $\exists \Gamma'''$ s.t. $\Gamma \rightarrow \Gamma'''$ with $\Gamma''' \leqslant \Gamma''$.*

Proof. If $\Gamma' \rightarrow \Gamma''$, then the reduction is a result from the communication between some two entries $s[p] : S'_\oplus, s[q] : S'_\&$ in Γ'.

Since $\Gamma \leqslant \Gamma'$, then $\Gamma = \Gamma_0 \cdot \Gamma_\rho, s[p] : S_\oplus, s[q] : S_\&$, where Γ_ρ only contains singletons (and hence $\mathsf{end}(\Gamma_\rho)$).

By lemma 15, we know $\Gamma \rightarrow \Gamma'''$. Lastly, we obtain $\Gamma''' \leqslant \Gamma''$ since *(i)* for entries $s[p]$ and $s[q]$, we observe from definition 2 that type continuations preserve subtyping; and *(ii)* the remainder of the context is unaffected by the communication. ◻

B.6 Session Inversion

Lemma 17. *If $P \equiv \mathbf{0}$ and $\Theta ; \Gamma \vdash P$, then $\mathsf{end}(\Gamma)$.*

Proof. By lemma 13 $\exists \Gamma' \equiv \Gamma$ s.t. $\Theta ; \Gamma' \vdash \mathbf{0}$. By [T-**0**], $\mathsf{end}(\Gamma')$. By definition 5, we obtain $\mathsf{end}(\Gamma)$.

Lemma 18. *Assume $\emptyset ; \Gamma \vdash \big|_{q \in I} P_q$ with each P_q either being $\mathbf{0}$ (up-to $\equiv$) or only playing role q in s. Then, $\Gamma = \Gamma_0, \{s[q] : U_q\}_{q \in I'}$ (for some I') and with $\mathsf{end}(\Gamma_0)$. Moreover, $\forall q \in I'$:*

1. *if $!\rho \&_{j \in J} \mathsf{m}_j(\widetilde{T_j}).S_j \leqslant U_q$ then $q \in I$ and for some $\mathbb{C}, \mathbb{C}'$, and $J' \supseteq J$, either:*
   - $P_k \equiv \mathbb{C}\left[!s[q][\rho]\&_{j \in J'} \mathsf{m}_j(\widetilde{b_j}).P'_j\right]$ *or*
   - $P_k \equiv \mathbb{C}\left[\begin{array}{l}\mathsf{def}\ \mathrm{X}(x_1 : S'_1, \ldots, x_n : S'_n) = \mathbb{C}'\left[!x_l[\rho]\&_{j \in J'} \mathsf{m}_j(\widetilde{b_j}).P'_j\right] \mathsf{in} \\ \quad \mathrm{X}\left\langle s'_1[r_1], \ldots, s'_{l-1}[r_{l-1}], s[q], s'_{l+1}[r_{l+1}], \ldots, s'_n[r_n]\right\rangle\end{array}\right]$ *with $1 \leq l \leq n$;*
2. *if $\rho \&_{j \in J} \mathsf{m}_j(\widetilde{T_j}).S_j \leqslant U_q$ then $q \in I$ and for some $\mathbb{C}, \mathbb{C}'$, and $J' \supseteq J$, either:*
   - $P_k \equiv \mathbb{C}\left[!s[q][\rho]\&_{j \in J'} \mathsf{m}_j(\widetilde{b_j}).P'_j\right]$ *or*
   - $P_k \equiv \mathbb{C}\left[\begin{array}{l}\mathsf{def}\ \mathrm{X}(x_1 : S'_1, \ldots, x_n : S'_n) = \mathbb{C}'\left[!x_l[\rho]\&_{j \in J'} \mathsf{m}_j(\widetilde{b_j}).P'_j\right] \mathsf{in} \\ \quad \mathrm{X}\left\langle s'_1[r_1], \ldots, s'_{l-1}[r_{l-1}], s[q], s'_{l+1}[r_{l+1}], \ldots, s'_n[r_n]\right\rangle\end{array}\right]$ *with $1 \leq l \leq n$;*
3. *if $\oplus_{j \in J} \rho_j \mathsf{m}_j(\widetilde{T_j}).S'_j \leqslant U_q$ then $q \in I$ and for some $\mathbb{C}, \mathbb{C}'$ and $J' \subseteq J$, either:*
   - $P_k \equiv \mathbb{C}\left[\sum_{j \in J'} s[q][\rho_j] \oplus \mathsf{m}_j\langle\widetilde{V_j}\rangle.P'_j\right]$ *or*
   - $P_k \equiv \mathbb{C}\left[\begin{array}{l}\mathsf{def}\ \mathrm{X}(x_1 : S'_1, \ldots, x_n : S'_n) = \mathbb{C}'\left[\sum_{j \in J'} x_l[\rho_j] \oplus \mathsf{m}_j\langle\widetilde{V_j}\rangle.P'_j\right] \mathsf{in} \\ \quad \mathrm{X}\left\langle s'_1[r_1], \ldots, s'_{l-1}[r_{l-1}], s[q], s'_{l+1}[r_{l+1}], \ldots, s'_n[r_n]\right\rangle\end{array}\right]$ *with $1 \leq l \leq n$.*
4. *if $S_1 \mid \cdots \mid S_n \leqslant U_q$ then $q \in I$ and $P_q \equiv P_1 \mid \cdots \mid P_n$ s.t. $\forall j \in 1..n : \emptyset ; \Gamma_0, \{s[q] : S_j\} \vdash P_j$, with each P_j either being $\mathbf{0}$ (up-to $\equiv$) or only plays q in s.*

Further, 5. $\forall q \in I \setminus I' : P_q \equiv \mathbf{0}$.

Proof. Items *1–3* and *5* follow similar proofs to the base version of this theory, found in [25, Theorem B.4]. (Main differences being that for *3* we infer the shape of the choice before the selection; and the proof for *1* is new but almost identical to that of *2*.) Item *4* follows from the definition of definition 7.

C Subject Reduction and Session Fidelity

Theorem 1 *If $\Theta ; \Gamma \vdash P$ with Γ safe. Then, $P \to P'$ implies $\exists \Gamma'$ safe s.t. $\Gamma \to^* \Gamma'$ and $\Theta ; \Gamma' \vdash P'$.*

Proof. The proof is by induction on the derivation of $P \to P'$. Cases [R-C], [R-!C$_1$], [R-!C$_2$] are similar. For each of these we infer the shape of Γ by inversion of typing rules, observing that Γ reduces via communication actions, and rebuilding a typing derivation for the reduced process using the reduced context. Cases [R-+] and [R-X] are straight forward. Case [R-$\equiv$] holds from subject congruence (lemma 13), and case [R-$\mathbb{C}$] holds by a further proof by induction on the structure of $\mathbb{C}$.

We demonstrate the proof with case [R-!C$_2$]:
We may assume: **(A1)** $\Theta ; \Gamma \vdash P$ **(A2)** $\varphi(\Gamma)$ **(A3)** $P \to P'$

From the hypothesis and [R-!C$_2$], we infer the shape of P and P':

$$P = s[q][p] \oplus \mathsf{m}_k\langle\widetilde{d}\rangle \;.\; Q \;\mid\; !s[p][\alpha] \;\&_{i\in I}\; \mathsf{m}_i(\widetilde{b_i}) \;.\; Q'_i \qquad \text{(by rule [R-!C}_2\text{])} \quad (20)$$

$$P' = Q \;\mid\; !s[p][\alpha] \;\&_{i\in I}\; \mathsf{m}_i(\widetilde{b_i}) \;.\; Q'_i \;\mid\; Q'_k\{\widetilde{d}/\widetilde{b_k}\}\{q/\alpha\} \qquad \text{(by rule [R-!C}_2\text{])} \quad (21)$$

$$k \in I \qquad \text{(by rule [R-!C}_2\text{])} \quad (22)$$

We can now infer the shape of Γ through inversion of typing rules:

$$\dfrac{\begin{array}{c}\Gamma_L = \Gamma_{L_0} \cdot \Gamma_\oplus \cdot \Gamma_\rho(\,\cdot\,\Gamma_{L_i})_{i\in 1..n} \vdash s[q][p] \oplus \mathsf{m}_k\langle d_1,\ldots,d_n\rangle \;.\; Q \\ \Gamma_R = \Gamma_{R_0} \cdot \Gamma_! \vdash !s[p][\alpha] \;\&_{i\in I}\; \mathsf{m}_i(\widetilde{b_i}) \;.\; Q'_i\end{array}}{\Gamma = \Gamma_L \cdot \Gamma_R \vdash s[q][p] \oplus \mathsf{m}_k\langle\widetilde{d}\rangle \;.\; Q \;\mid\; !s[p][\alpha] \;\&_{i\in I}\; \mathsf{m}_i(\widetilde{b_i}) \;.\; Q'_i}\;[\text{T-}|]$$

$$\dfrac{\Gamma_\oplus \vdash s[q] : p \oplus m_k(T_1,\ldots,T_n).S_L \qquad \Gamma_\rho \vdash p : p \qquad \forall i \in 1..n : \Gamma_{L_i} \vdash d_i : T_i \qquad \Gamma_{L_0} + s[q] : S_L \vdash Q}{\Gamma_{L_0} \cdot \Gamma_\oplus \cdot \Gamma_\rho(\,\cdot\,\Gamma_{L_i})_{i\in 1..n} \vdash s[q][p] \oplus \mathsf{m}_k\langle d_1,\ldots,d_n\rangle \;.\; Q}\;[\text{T-}\oplus]$$

$$\dfrac{\Gamma_! \vdash s[p] : !\alpha\&_{i\in I} m_i(\widetilde{T'}).S_{R_i} \qquad \mathsf{end}(\Gamma_{R_0}) \qquad \left(\Gamma_{R_0} + s[p] : S_{R_i} + \widetilde{b_i} : \widetilde{T_i} \hookleftarrow \alpha \vdash Q'_i\right)_{i\in I}}{\Gamma_{R_0} \cdot \Gamma_! \vdash !s[p][\alpha] \;\&_{i\in I}\; \mathsf{m}_i(\widetilde{b_i}) \;.\; Q'_i}\;[\text{T-!}]$$

Since Γ is *safe*, we now that:

$$|\,\widetilde{T}\,| = |\,\widetilde{T'}\,| \qquad \text{(by condition [S-!}\oplus\&\text{])} \quad (23)$$

$$\forall j \in 1..|\,\widetilde{T}\,| : T_j \leqslant T'_j \;\text{ if } T_j \text{ is a session type} \qquad \text{(by condition [S-!}\oplus\&\text{])} \quad (24)$$

Without loss of generality, we assume $\widetilde{T}$ and $\widetilde{T'}$ to be ordered as session types first, followed by role singletons, s.t. $\widetilde{T} = \widetilde{S'}, \widetilde{r}$ and $\widetilde{T'} = \widetilde{S''}, \widetilde{\kappa}$. Now we observe the context reduction using (23),(24), lemma 16 and [Γ-Com$_2$]:

$$\Gamma_\oplus, \Gamma_! \to \Gamma'' \leqslant \Gamma_0, \{s[q] : S_L + s[p] : !\alpha\&_{i\in I} m_i(\widetilde{T'_i}).S_{R_i} \mid S_{R_k}\{q/\alpha\}\{\widetilde{r}/\widetilde{\kappa}\}\} \qquad \text{(by [}\Gamma\text{-Com}_2\text{])} \quad (25)$$

$$\Gamma'' = (\Gamma'_L \leqslant \Gamma_0, s[q] : S_L) \cdot \Gamma_! \cdot (\Gamma'_R \leqslant \Gamma_0, s[p] : S_{R_k}\{q/\alpha\}\{\widetilde{r}/\widetilde{\kappa}\}) \qquad \text{(by (25) and figure 5)} \quad (26)$$

where Γ_0 contains any singletons in $\Gamma_\oplus, \Gamma_!$, and therefore, $\mathsf{end}(\Gamma_0)$.

Using rule [Γ-Cong], we can infer the shape of the entire context reduction:

$$\Gamma \to \Gamma' = (\Gamma_{L_0} \cdot \Gamma_\rho(\,\cdot\,\Gamma_{L_i})_{i\in 1..n} \cdot \Gamma_{R_0}) + (\Gamma'') \qquad \text{(by (25),(26) and [}\Gamma\text{-Cong])} \quad (27)$$

$$\Gamma' = (\Gamma_{R_0} \cdot \Gamma_!) \cdot (\Gamma_{L_0} \cdot \Gamma'_L) \cdot (\Gamma'_R(\,\cdot\,\Gamma_{L_i})_{i\in 1..n}) \qquad \text{(by (26) and (27))} \quad (28)$$

where Γ_ρ is engulfed in Γ'_R since Γ'_R contains Γ_0, which contains all singletons that exist in $\Gamma_\oplus, \Gamma_\&$.

We can now observe that $\Gamma' \vdash P'$ since:

$$\Gamma_{R_0} \cdot \Gamma_! \vdash !s[p][\alpha] \;\&_{i\in I}\; \mathsf{m}_i(\widetilde{b_i}) \;.\; Q'_i \qquad \text{(from der. tree)} \quad (29)$$

$$\Gamma_{L_0} \cdot \Gamma'_L \vdash Q \qquad \text{(from der. tree, (26) and lemma 12)} \quad (30)$$

$$\Gamma'_R(\,\cdot\,\Gamma_{L_i})_{i\in 1..n} \vdash Q'_k\{\widetilde{d}/\widetilde{b_k}\}\{q/\alpha\} \qquad \text{(from der. tree, (26), lemma 12, lemma 5 and lemma 6)} \quad (31)$$

$$\Gamma' \vdash P' \qquad \text{(by (29),(30),(31))} \quad (32)$$

Lastly, we must show that Γ' is *safe*, which holds from item **(A2)**, [S-$\to$] and (27). ◻

Theorem 2 *Assume* $\emptyset\,;\Gamma \vdash P$ *with* Γ *safe and* $P \equiv \prod_{q\in I} P_q$ *where each* P_q *is either* $\mathbf{0}$ *(up-to* $\equiv$*), or only plays role* q *in* s*. Then,* $\Gamma\rightarrow$ *implies* $\exists \Gamma', P'$ *s.t.* $\Gamma\rightarrow\Gamma'$*,* $P \rightarrow^* P'$ *and* $\Gamma' \vdash P'$*, where* $P' \equiv \prod_{q\in I} P_q{}'$ *and each* $P_q{}'$ *is either* $\mathbf{0}$ *(up-to* $\equiv$*), or only plays role* q *in* s*.*

Proof (Sketch). We start by observing that if $\varphi(\Gamma)$ and $\Gamma\rightarrow$, then we can split out of Γ either *(i)* a selection and branch type; or *(ii)* a selection and replicated branch type. Proceeding in a similar manner for both cases, we infer the shape of P based on the split types. It follows that P can reduce via [R-C] (or [R-!C_1]/[R-!C_2] in case *(ii)*), possibly after some applications of [R-$\mathbb{C}$],[R-X], and [R-+]; and Γ can reduce such that the process reduction remains typable. Lastly, we show that the process reductum retains the structure of the assumptions for session fidelity (by rule [R-$\equiv$] and since Definition 7 requires its properties to hold on all subterms). We note that for replicated communication, a process P_p can take the shape $P_p \equiv \prod_{i\in I} P'_i$ where each P'_i is either $\mathbf{0}$ (up-to $\equiv$), or only plays role p in s; *i.e.*, a process playing a single role can consist of multiple parallel processes that all play the same role. In this case we first show that P_p is typed under a runtime type U, then proceed in a similar manner as before.

D Unfolding

Below we present our definitions for type and context unfolding; they are mostly standard, we only adapt type unfolding to cater for parallel types.

Definition 12 (Type unfolding). *The one-step unfolding of a type* U*, written* $\mathsf{unf}(U)$*, is given by:*

$$\mathsf{unf}(\mu \mathrm{t}.S) = S\{\mu \mathrm{t}.S/\mathrm{t}\} \qquad \mathsf{unf}(S \mid U) = \mathsf{unf}(S) \mid \mathsf{unf}(U) \qquad \mathsf{unf}(S) = S \quad \textit{if } S \neq \mu \mathrm{t}.S'$$

The n-steps unfolding of a type U*, written* $\mathsf{unf}^n(U)$*, is given by:*

$$\mathsf{unf}^0(U) = U \qquad \mathsf{unf}^{m+1}(U) = \mathsf{unf}(\mathsf{unf}^m(U))$$

The complete unfolding of a type U*, written* $\mathsf{unf}^*(U)$*, is defined as:*

$$\mathsf{unf}^*(U) = \mathsf{unf}^n(U) \quad \textit{for the smallest } n \textit{ s.t. } \mathsf{unf}^n(U) = \mathsf{unf}^{n+1}(U)$$

Definition 13 (Context unfolding, as in [25] definition K.1). *The set of unfoldings of a type context* Γ*, written* $\mathsf{unf}(\Gamma)$*, is defined below (where* $\Gamma\{U/c\}$ *is a mapping update):*

$$\mathsf{unf}(\Gamma) = \bigcup_{c\,:\,U\in\Gamma}\{\Gamma\{\mathsf{unf}(U)/c\}\} \qquad \textit{extends to sets of contexts as} \qquad \mathsf{unf}(\{\Gamma_i\}_{i\in I}) = \bigcup_{i\in I}\mathsf{unf}(\Gamma_i)$$

Given a set of contexts ξ*, the closure of its unfoldings, written* $\mathsf{unf}*(\xi)$*, is defined as:*

$$\mathsf{unf}^*(\xi) = \mathsf{lfix}(\lambda\xi'.\xi \cup \xi' \cup \mathsf{unf}(\xi \cup \xi')) \quad \textit{where } \mathsf{lfix} \textit{ is the least fixed point of its argument}$$