Mixed Choice Multiparty Session Types, Precisely

Jake Masters and Nobuko Yoshida

University of Oxford, UK
{jake.masters,nobuko.yoshida}@cs.ox.ac.uk

Abstract. A *precise* (sound and complete) subtyping relation $\leqslant$ specifies that T' is a subtype of T if and only if a program of type T' can always safely replace a program of type T without compromising the safety of a larger program. This paper formulates and proves preciseness of subtyping for *mixed choice multiparty session types* with *session delegation*, *creation*, and *interleaving*. We prove soundness by developing the first general type system for a full mixed choice multiparty session π-calculus. To prove completeness, we introduce the *three-party lock*, which is a minimal and general form of liveness error for handling interleaved sessions, and we establish a new proof technique based on a construction of *scheduler processes* which enable exhaustive detection for all failures of the subtyping relation. We then extend the preciseness results to a family of mixed choice multiparty session types. Algorithms for checking (1) subtyping and (2) safety, deadlock-freedom, and liveness of typing contexts are fully implemented and optimised to run in quadratic time with respect to the size of the state space and typing context, and have been evaluated with (mixed choice) case studies from the literature.

1 Introduction

Precise Subtyping. Liskov and Wing's substitution principle [40] defines the notion of *subtyping* by specifying when one type safely substitutes for another, i.e., if type T' is a subtype of T (denoted by $T' \leqslant T$), then a program P of type T' can always be used in place of P' of type T. This principle is integrated into a typing system usually with a *subsumption rule*: if program P is typed by T' and $T' \leqslant T$, then P is typed by T. We then wish our type system to be *maximally expressive*, while remaining *sound*. Larger subtyping relations empower the subsumption rule, enlarging the set of typable programs, but adding too many relations to the subtyping relation makes the type system unsound. For instance, if we let $\mathsf{bool} \leqslant \mathsf{int}$, then we mistakenly type the expression $\mathtt{true} + 5$, which fails to evaluate and causes a run-time error.

For a type system ($\vdash$), write $\mathbb{C}_T$ for a context such that $\{X{:}\{c : T\}\} \vdash \mathbb{C}_T[X] : \emptyset$ for channel c and variable X. Subtyping relation ($\leqslant$) is *precise* if:

1. (*sound*) using $\leqslant$ for subsumption ensures that typed processes are correct: $T' \leqslant T$ implies that if $\vdash P : \{c : T'\}$, then $\mathbb{C}_T[P]$ does not behave badly;

2. (*complete*) it cannot be extended without becoming unsound: $T' \not\leqslant T$ implies that for some $\mathbb{C}_T$ and P, we have $\vdash P : \{c : T'\}$ and $\mathbb{C}_T[P]$ behaves badly.

Preciseness is used to *test* the canonicity of subtyping with respect to the operational semantics of a target language. The problem of determining precise subtyping relations has been widely studied for various typed λ-calculi [4,56,33,58,20,39], and later for *session types* [28,27,54], where subtyping plays an essential role in their integration into programming languages, beginning with the initial work on Java [32]. Preciseness was first studied for synchronous and asynchronous binary session types [13] for a calculus with session delegations.

Mixed Choice Multiparty Session Types. *Multiparty Session Types* (MPST) [29,30,60,15] are a typing disciple for message passing processes that are distributed across multiple participants. Typing ensures adherence to a specified protocol, used to guarantee *communication safety* (type and choice safety), *deadlock-freedom* (no process becomes stuck during execution) and *liveness* (a communication, which is ready, eventually happens) of concurrent and distributed systems. MPST has been integrated in over 26 programming languages, adapted to static or dynamic checking, type inference and code generation [59].

An extension of MPST, *mixed choice multiparty session types* [48] provide *non-deterministic choices* between sets of inputs and outputs directed to multiple participants. Mixed choice in MPST *strictly* increases the expressivity of typable protocols [48]. Although the original π-calculus [43] included non-deterministic choice, the study of mixed choice MPST is a recent development, and its need is growing. Recent work on mixed choice multiparty session types include probability [7], modelling federated learning protocols [51], automata-based analyses [21], and integration into a functional language [26] and Erlang [8].

Precise Mixed Choice Multiparty Session Subtyping. Preciseness of subtyping has been studied for synchronous and asynchronous MPST [23,24], but two constructions remain untreated: (1) [23,24] are limited to calculi without session delegation, creation, or interleaving; and (2) [23,24] do not treat mixed choice. This paper tackles these two key constructions, determining the precise subtyping relation $\leqslant$ for mixed choice multiparty synchronous session types with session delegation, creation, and interleaving. We consider the subtyping first presented in [48] and its restrictions to a family of subcalculi.

While soundness immediately follows from type safety, proving completeness with these two extensions is non-trivial since the previous methods in [13,23,24] are not applicable. To explain this problem, we give a general strategy (in a contrapositive form) for proving completeness.

- **[Step 1]** For each type T, define a complementary typing context Δ_T to schedule the execution of T.
- **[Step 2]** Define a T-typed *characteristic process* P_T and a *characteristic context* $\mathbb{C}_T$ which behaves well when filled with T-typed processes.
- **[Step 3]** Characterise the negation of the subtyping relation.
- **[Step 4]** Show that if $T' \not\leqslant T$, then $\mathbb{C}_T[P]$ behaves badly.

The first challenge is to define what "behaving badly" means. In the binary case [13], simple error rules are sufficient because each characteristic process is composed with a characteristic context which has a dual channel type. "Bad behaviours" in [23,24] are "deadlock" and "livelock" for a *single* multiparty session, neither of which are useful for a full session π-calculus since there is no standard typing system which can ensure deadlock-freedom or liveness for interleaved sessions. In this paper, we solve this issue by introducing a minimal but general liveness error, called a *three-party lock*. Here "minimal" refers to the number of participants engaging with a lock, and "general" means that it captures all livelock patterns induced by the complementary typing context's scheduling. Thus interactions between characteristic processes and contexts of incompatible types, induce noticeably "bad behaviour". A three-party lock occurs when each participant is prevented from communicating with another participant due to circular dependency. Process P below is a simple example of a three-party lock: $P = s[\mathtt{p}_0][\mathtt{p}_1]?(x).\mathbf{0} \mid s[\mathtt{p}_1][\mathtt{p}_2]?(x).\mathbf{0} \mid s[\mathtt{p}_2][\mathtt{p}_0]?(x).\mathbf{0}$ where $s[\mathtt{p}]$ denotes session s with participant $\mathtt{p}$ and ? denotes input. In P, $\mathtt{p}_0$ is dependent on $\mathtt{p}_1$ to communicate, $\mathtt{p}_1$ on $\mathtt{p}_2$, and $\mathtt{p}_2$ on $\mathtt{p}_0$. Later we prove that the 2-party lock used in [13] is *insufficient* as an error to prove completeness for multiparty mixed choice, highlighting the difference between binary and multiparty.

The second challenge is to define characteristic processes and contexts. Mixed choice allows subtyping to fail in many more ways than with separate choice (where choice is either input or output) in [13,23,24]. Resultantly, the size of characteristic contexts grows exponentially in the size of their types. To capture these failures and avoid the complexity of reasoning, we build *complementary contexts*, $\mathbb{A}[s, T]$ for each type T, with a *scheduler process* to monitor all possible failures of subtyping non-deterministically. The scheduler keeps track of the expected shape of T during execution, and non-deterministically controls the other participants to execute T and test for incorrect behaviours, forcing errors and three-party locks if any are found. We use complementary contexts to construct characteristic processes and contexts (see Ex. 4.1 and Ex. 4.2).

Contributions. In this work, we study the preciseness of the mixed choice multiparty subtyping relation $\leqslant$ (Def. 3.2). First, we introduce *three-party locks* for both processes (Def. 2.4) and typing contexts (Def. 3.5). We show that $\leqslant$ in [48] is a canonical subtyping for mixed choice multiparty session types, proving *operational* (Thm. 4.6) and *denotational preciseness* (Thm. 4.7) with respect to *safety* and *three-party lock-freedom* in the mixed choice multiparty π-calculus (Def. 2.1). This is the first preciseness result involving, not only mixed choice, but also session delegation in a multiparty session π-calculus.

We base our typing system on the bottom-up approach [52] instead of using the top-down approach starting from global protocols [23]. A key technique to prove completeness is constructing *complementary contexts* (Def. 4.2), which exhaustively check for the greater variety of failures of subtyping.

Our approach is generalised to a family of calculi defined in [48] (Def. 5.3 and Thm. 5.5), including the calculus in [23] (Thm. 5.4), demonstrating extensibility of our approach to various MPST calculi with/without mixed choice.

Further, we provide new and efficient algorithms with implementations, to decide safety, deadlock-freedom, and liveness of typing contexts. Their complexities are quadratic in the state space and context, an improvement over the exponential algorithm [55, Theorem 6.20] and model checking approach [52, § 6].

We have identified and corrected small but crucial errors in several existing work [48,52,23]. Additionally, to our best knowledge, this paper offers the first typing system which guarantees communication safety, deadlock-freedom and liveness in the presence of mixed choice and session delegations.

Full proofs and further explanations are found in [42]. An implementation of the algorithms and benchmarks in § 6.1 is provided in the repository [1].

2 Mixed Choice Multiparty Session Calculus

This section defines the mixed choice multiparty session π-calculus ($\mathsf{MCM}\pi$), extending the calculus with delegation and hiding in [52] with mixed choice [48]. After defining properties of processes, we introduce the notion of a *three-party lock* for processes, which plays a key role in the proof of preciseness.

Syntax and Semantics. We first define the syntax of the $\mathsf{MCM}\pi$-calculus.

Definition 2.1 (Mixed Choice Multiparty π-Calculus). The syntax of the *mixed choice multiparty π-calculus* ($\mathsf{MCM}\pi$) is defined by:

$$
\begin{array}{lll}
v ::= x \mid \mathtt{n} \mid \mathtt{true} \mid \mathtt{false} & & \textit{(variable, nat, bools)} \\
e ::= v \mid e = e' \mid e < e' \mid \mathtt{succ}(e) \mid \mathtt{neg}(e) & & \textit{(value, expressions)} \\
c ::= y \mid s[\mathtt{p}] & & \textit{(chan variable, chan with role } \mathtt{p}\textit{)} \\
\pi ::= c[\mathtt{p}]!\mathtt{m}\langle c'\rangle \mid c[\mathtt{p}]!\mathtt{m}\langle e\rangle \mid c[\mathtt{p}]?\mathtt{m}(y) \mid c[\mathtt{p}]?\mathtt{m}(x) & & \textit{(output to/input prefix from } \mathtt{p}\textit{)} \\
P ::= \mathbf{0} \mid (\nu s)\, P \mid X \mid \mu X.P \mid P \mid P' & & \textit{(nil, hiding, proc var, recursion, par)} \\
\quad \mid \mathsf{if}\, e\, \mathsf{then}\, P\, \mathsf{else}\, Q \mid \textstyle\sum_{i\in I} \pi_i.P_i \mid \mathtt{err} & & \textit{(sum with } I \neq \emptyset\textit{, error)}
\end{array}
$$

Values $(v, v', w, ...)$ are *expression variables* $(x, x', \dots)$, integers, or booleans. *Expressions* $(e, e', ...)$ are values, comparisons ($=$ and $<$), successors, or negations. *Channels* $(c, c', ...)$ are *channel variables* $(y, y', \dots)$ or channels with roles $s[\mathtt{p}]$, representing endpoints whose users plays role $\mathtt{p}$ in the session s. The *prefix* $c[\mathtt{p}]!\mathtt{m}\langle c'\rangle$ (resp. $c[\mathtt{p}]!\mathtt{m}\langle e\rangle$) denotes *sending* a message through *subject* c with label $\mathtt{m}$ towards role $\mathtt{p}$ with the channel c' (resp. the expression e) as the payload. The prefixes $c[\mathtt{p}]?\mathtt{m}(y)$ and $c[\mathtt{p}]?\mathtt{m}(x)$ denote *receiving* messages, dual to the above prefixes. $\mathsf{subj}(\pi)$ denotes the subject of π. Other constructors are standard.

In *processes* $(P, Q, R, ...)$, the only non-standard constructor is the *sum* constructor, which implements mixed choice. *Sum* $\sum_{i\in I} \pi_i.P_i$ denotes the nondeterministic choice of the actions $\{\pi_i\}_{i\in I}$ with continuation processes $\{P_i\}_{i\in I}$, where received payloads are substituted into the continuation. Given two sum processes, we often write their sum, $\sum_{i\in I} \pi_i.P_i + \sum_{i\in J} \pi_i.P_i = \sum_{i\in I\cup J} \pi_i.P_i$. We write $\dagger$ for ! or ? and $\overline{!} = ?$ and $\overline{?} = !$. *Session restriction* (hiding), $(\nu s)\, P$, creates a new session s with scope P. *Process variable*, X, and *recursive process*,

$$
\begin{array}{ll}
\text{[R-Val]} & e \downarrow v \implies (s[\mathtt{q}][\mathtt{p}]?\mathtt{m}(x).Q + Q') \mid (s[\mathtt{p}][\mathtt{q}]!\mathtt{m}\langle e\rangle.P + P') \to Q\{v/x\} \mid P \\
\text{[R-Chan]} & (s[\mathtt{q}][\mathtt{p}]?\mathtt{m}(y).Q + Q') \mid (s[\mathtt{p}][\mathtt{q}]!\mathtt{m}\langle s'[\mathtt{p}']\rangle.P + P') \to Q\{{}^{s'[\mathtt{p}']}/_{y}\} \mid P \\
\text{[R-Cond]} & e \downarrow v \ \wedge\ v \in \{\mathtt{true}, \mathtt{false}\} \implies \mathsf{if}\ e\ \mathsf{then}\ P_{\mathtt{true}}\ \mathsf{else}\ P_{\mathtt{false}} \to P_v \\
\text{[R-Ctx]} & P \Rrightarrow P' \ \wedge\ P' \to Q' \ \wedge\ Q' \Rrightarrow Q \implies \mathbb{C}[P] \to \mathbb{C}[Q] \\
\text{[E-Cond]} & e \downarrow v \ \wedge\ v \notin \{\mathtt{true}, \mathtt{false}\} \implies \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \to \mathtt{err} \\
\text{[E-Eval]} & e \downarrow \mathtt{err} \implies c[\mathtt{p}]!\mathtt{m}\langle e\rangle.P + P' \to \mathtt{err} \\
\text{[E-M-I]} & (s[\mathtt{q}][\mathtt{p}]?\mathtt{m}(x).Q + Q') \mid (s[\mathtt{p}][\mathtt{q}]!\mathtt{m}\langle c\rangle.P + P') \to \mathtt{err} \\
\text{[E-M-II]} & (s[\mathtt{q}][\mathtt{p}]?\mathtt{m}(y).Q + Q') \mid (s[\mathtt{p}][\mathtt{q}]!\mathtt{m}\langle e\rangle.P + P') \to \mathtt{err} \\
\text{[E-M-III]} & (s[\mathtt{p}][\mathtt{q}]!\mathtt{m} \prec P \ \wedge\ s[\mathtt{q}][\mathtt{p}]?\mathtt{m}' \prec Q \ \wedge\ s[\mathtt{q}][\mathtt{p}]?\mathtt{m} \not\prec Q) \implies P \mid Q \to \mathtt{err}
\end{array}
$$

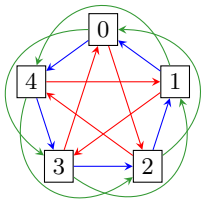
Fig. 1: Semantics of the Mixed Choice Multiparty π-Calculus

$\mu X.P$, represent recursion. We assume P is guarded, i.e., $\mu X.X$ is not allowed. Other processes are standard. $\mathtt{err}$ denotes the error process.

Restriction, branching, and recursive binders, act as binders. $\mathsf{fpv}/\mathsf{chan}$ is the *free process variable/channel* functions. We assume the Barendregt convention where all bound sessions and variables are assumed pairwise distinct, and distinct from any free sessions or variables. We assume the existence of capture-avoiding substitution operators on processes and expressions. We define *reduction contexts* by $\mathbb{C}[] ::= [] \mid (\nu s_1, \ldots, s_n)([] \mid P)$ and the *prefix* notation: $c[\mathtt{p}]!\mathtt{m} \prec c[\mathtt{p}]!\mathtt{m}\langle d\rangle$, $c[\mathtt{p}]?\mathtt{m} \prec c[\mathtt{p}]?\mathtt{m}(z)$, $c[\mathtt{p}]\dagger\mathtt{m} \prec \sum_{i\in I} \pi_i.P_i$ if $c[\mathtt{p}]\dagger\mathtt{m} \prec \pi_k$ for some $k \in I$, and similarly for $c[\mathtt{p}]\dagger$. We write $\pi \not\prec \pi'$ etc. if $\pi \prec \pi'$ is not true.

We now introduce our running example, the *leader election* protocol from [48, Example 2.2], extended with *session delegations*.

Example 2.1 (Leader Election with Delegation). Five participants elect a leader in three stages of voting. The first round is indicated by blue, the second by red, and the third by green. In round one, the ith participant either votes using the token $s_i[\mathtt{p}]$ or is voted for. In round two, if a participant is voted for, then it either votes or is voted for again. In round three, the participant with two votes receives a final vote from the remaining participant. After taking part in the election, the participants redeem their tokens.



$$
\begin{array}{rl}
P_i = & s[\mathtt{p}_i][\mathtt{p}_{(i+4)\%5}]!\mathtt{elect}\langle s_i[\mathtt{p}]\rangle + s[\mathtt{p}_i][\mathtt{p}_{(i+1)\%5}]?\mathtt{elect}(y).P_i'(y) + \\
& s[\mathtt{p}_i][\mathtt{p}_{(i+3)\%5}]!\mathtt{elect}\langle s_i[\mathtt{p}]\rangle \\
P_i'(c) = & s[\mathtt{p}_i][\mathtt{p}_{(i+2)\%5}]!\mathtt{elect}\langle s_i[\mathtt{p}]\rangle.c[\mathtt{s}]!\mathtt{token}\langle \mathtt{i}\rangle + \\
& s[\mathtt{p}_i][\mathtt{p}_{(i+3)\%5}]?\mathtt{elect}(y').s[\mathtt{p}_i][\mathtt{p}_{(i+2)\%5}]?\mathtt{elect}(y'').P_i''(c, y', y'') \\
P_i''(c, c', c'') = & s_i[\mathtt{p}][\mathtt{s}]!\mathtt{token}\langle \mathtt{i}\rangle.c[\mathtt{s}]!\mathtt{token}\langle \mathtt{i}\rangle.c'[\mathtt{s}]!\mathtt{token}\langle \mathtt{i}\rangle.c''[\mathtt{s}]!\mathtt{token}\langle \mathtt{i}\rangle \ (0 \le i \le 4) \\
Q = & \Pi_{0\le i\le 4} s_i[\mathtt{s}][\mathtt{p}]?\mathtt{token}(x) \qquad P_{\text{lead}} = P_0 \mid P_1 \mid P_2 \mid P_3 \mid P_4 \mid Q
\end{array}
$$

Definition 2.2 (Operational Semantics). We use a reflexive and transitive *structural precongruence* on processes, written $P \Rrightarrow P'$, defined by the standard

rules, including $\mu X.P \Rrightarrow P\{^{\mu X.P}/_X\}$. We write $P \equiv P'$ if $P \Rrightarrow P'$ and $P' \Rrightarrow P$. *Reduction rules* on processes, $\rightarrow$, are defined by the rules in Fig. 1 where we assume the standard expression evaluation relation, $e \downarrow v$ from [23]. We say that P *has an error* iff $P \Rrightarrow \mathbb{C}[\mathtt{err}]$ for some context $\mathbb{C}[]$. We define reduction of contexts, written $\mathbb{C}[] \rightarrow \mathbb{C}'[]$ iff $\forall P\colon \mathbb{C}[P] \rightarrow \mathbb{C}'[P]$. We define $\rightarrow^*$ to be the reflexive and transitive closure of $(\rightarrow \cup \Rrightarrow)$. $P \rightarrow$ denotes $\exists P'.\ P \rightarrow P'$.

Most rules are standard [61]. [R-Val] and [R-Chan] allow for the communication of values and channels across parallel components, with the received payload being substituted into the receiver's continuation. [E-Cond] says that conditioning on a non-boolean value causes an error. Similarly, [E-Eval] says that expression errors are errors. [E-M-I] (resp. [E-M-II]) says that using a channel (resp. value) as the payload when a value (resp. channel) must be received causes an error. [E-M-III] is the standard label mismatch error; it says that if two endpoints can communicate but the sent label is not expected, then an error occurs.

Definition 2.3 (Properties). Let P be a process. We say that P is:

- deadlocked iff $P \not\rightarrow$ and $P \not\Rrightarrow \mathbf{0}$; and
- livelocked/has a livelock iff $P = \mathbb{C}\left[\sum_{i\in I} \pi_i.P_i\right]$ and it is not the case that: $\mathbb{C}[\,] \rightarrow^* \mathbb{C}'\left[[\,] \mid \sum_{j\in J} \pi'_j.Q_j\right]$ and there exist s, $\mathtt{p}$, $\mathtt{q}$, $\dagger$, $\mathtt{m}$, $i \in I$, and $j \in J$ such that $s[\mathtt{p}][\mathtt{q}]\dagger\mathtt{m} \prec \pi_i$ and $s[\mathtt{q}][\mathtt{p}]\overline{\dagger}\mathtt{m} \prec \pi'_j$.
- *safe* iff for all $P \rightarrow^* P'$, P' has no error;
- *deadlock-free* iff for all $P \rightarrow^* P'$, P' is not deadlocked; and
- *live* iff for all $P \rightarrow^* P'$, P' is not livelocked.

A process is *deadlock-free* if it only stops reducing at $\mathbf{0}$. Process $s[\mathtt{p}][\mathtt{q}]!\mathtt{m}\langle 5\rangle.\mathbf{0}$ is deadlocked but safe. Liveness means that if P is waiting to perform an action, the rest of processes always move to a point where the action in P can be performed. All deadlocked processes are livelocked. Process $s[\mathtt{r}][\mathtt{p}]?\mathtt{m}(x).\mathbf{0} \mid \mu X.(s[\mathtt{q}][\mathtt{p}]?\mathtt{m}(x).X) \mid \mu X.(s[\mathtt{p}][\mathtt{q}]!\mathtt{m}\langle 0\rangle.X)$ is deadlock-free, but not live.

Three-Party Locks (3-plocks) are small and detectable livelocks. Intuitively, $(\nu s)\,Q$ has a 3-plock if: exactly one component can use $s[\mathtt{p}]$; it is being used in a choice; and every endpoint that it could communicate with, must first communicate with another endpoint, which must first communicate with $s[\mathtt{p}]$. This ensures that the endpoint $s[\mathtt{p}]$ can perform no actions.

Definition 2.4 (Three-Party Lock (3-plock)). We define the predicate $\mathsf{top}(P, s[\mathtt{p}_1][\mathtt{p}_2]\dagger)$ by the inductive rules:

$$\frac{}{\mathsf{top}(\mathbf{0}, s[\mathtt{p}_1][\mathtt{p}_2]\dagger)} \qquad \frac{}{\mathsf{top}(X, s[\mathtt{p}_1][\mathtt{p}_2]\dagger)} \qquad \frac{\mathsf{top}(Q, s[\mathtt{p}_1][\mathtt{p}_2]\dagger)}{\mathsf{top}((\nu s')\,Q, s[\mathtt{p}_1][\mathtt{p}_2]\dagger)}$$

$$\frac{}{\mathsf{top}(s[\mathtt{p}_1][\mathtt{p}_2]?\mathtt{m}(z).Q, s[\mathtt{p}_1][\mathtt{p}_2]?)} \qquad \frac{}{\mathsf{top}(s[\mathtt{p}_1][\mathtt{p}_2]!\mathtt{m}\langle w\rangle.Q, s[\mathtt{p}_1][\mathtt{p}_2]!)}$$

$$\frac{\mathsf{top}(Q, s[\mathtt{p}_1][\mathtt{p}_2]\dagger) \quad c \neq s[\mathtt{p}_1]}{\mathsf{top}(c[\mathtt{q}]?\mathtt{m}(z).Q, s[\mathtt{p}_1][\mathtt{p}_2]\dagger)} \qquad \frac{\mathsf{top}(Q, s[\mathtt{p}_1][\mathtt{p}_2]\dagger) \quad c \neq s[\mathtt{p}_1] \quad w \neq s[\mathtt{p}_1]}{\mathsf{top}(c[\mathtt{p}_3]!\mathtt{m}\langle w\rangle.Q, s[\mathtt{p}_1][\mathtt{p}_2]\dagger)}$$

$$\frac{\mathsf{top}(Q, s[\mathtt{p}_1][\mathtt{p}_2]\dagger)}{\mathsf{top}(\mu X.Q, s[\mathtt{p}_1][\mathtt{p}_2]\dagger)} \qquad \frac{\mathsf{top}(Q_i, s[\mathtt{p}_1][\mathtt{p}_2]\dagger) \ \ (i = 1, 2)}{\mathsf{top}(Q_1 \mid Q_2, s[\mathtt{p}_1][\mathtt{p}_2]\dagger)} \qquad \frac{\forall i \in I \quad \mathsf{top}(\pi_i.Q, s[\mathtt{p}_1][\mathtt{p}_2]\dagger)}{\mathsf{top}\left(\sum_{i\in I} \pi_i.Q, s[\mathtt{p}_1][\mathtt{p}_2]\dagger\right)}$$

We say that $\mathbb{C}[(\nu s)\, Q]$ has a *three-party lock (3-plock)* from $s[\mathtt{p}]$ if $Q = \sum_{i\in I} \pi_i.Q_i | P$, $s[\mathtt{p}] \notin \mathsf{chan}(P)$, and there are $\mathtt{q}_i$, $\mathtt{r}_i$ and $\dagger_i'$ for $i \in I$ such that $s[\mathtt{p}][\mathtt{q}_i]\dagger_i \prec \pi_i$, $\mathsf{top}(Q, s[\mathtt{q}_i][\mathtt{r}_i]\dagger_i')$, and: (1) if $\mathtt{r}_i = \mathtt{p}$ then $\dagger_i' \neq \overline{\dagger_i}$; and (2) if $\mathtt{r}_i \neq \mathtt{p}$ then there is $\dagger_i''$ such that $\mathsf{top}(Q, s[\mathtt{r}_i][\mathtt{p}]\dagger_i'')$.

We say that P is 3-plock free iff for all $P \to^* P'$, P' has no 3-plock.

If $s[\mathtt{p}_1] \notin \mathsf{chan}(P)$ then $\mathsf{top}(P, s[\mathtt{p}_1][\mathtt{p}_2]\dagger)$ is vacuously true. All 3-plocks are livelocks, but not all livelocks are 3-plocks. P_{lead} (Ex. 2.1) is 3-plock free. This is verified in Ex. 3.7 using typing. Chen et al. [13] introduced similar error conditions for an input/output session calculus without mixed choice, which we generalise to *two-participant locks (2-plocks)*. The negative result Thm. 4.3 shows that 2-plocks are insufficient to define 'bad behaviour' for multiparty sessions.

Example 2.2 (Three-Party Lock). Consider the following processes:

$$R_1 = (\nu s)\, s[\mathtt{p}][\mathtt{q}]!\mathtt{m}\langle 1\rangle \quad R_2 = (\nu s)\,(s[\mathtt{p}][\mathtt{q}]!\mathtt{m}\langle 2\rangle \mid s[\mathtt{q}][\mathtt{p}]!\mathtt{m}\langle 2\rangle)$$

$$R_3 = (\nu s)\,(s[\mathtt{p}][\mathtt{q}]!\mathtt{m}\langle 3\rangle \mid s[\mathtt{q}][\mathtt{r}]?\mathtt{m}(x).\mathbf{0} \mid s[\mathtt{r}][\mathtt{p}]!\mathtt{m}\langle 3\rangle)$$

R_1, R_2, and R_3 all have 3-plocks: R_1 because $s[\mathtt{p}]$ wants to communicate with $s[\mathtt{q}]$, which is not present; R_2 because $s[\mathtt{p}]$ wants to send a message to $s[\mathtt{q}]$, which must first send a message to $s[\mathtt{p}]$; and R_3 because $s[\mathtt{p}]$ wants to communicate with $s[\mathtt{q}]$, which must first communicate with $s[\mathtt{r}]$, which must first communicate with $s[\mathtt{p}]$. R_1, R_2 and R_3 are deadlocked, but three participant locks can be deadlock-free: e.g. for $R \in \{R_1, R_2, R_3\}$, $R \mid \mu X.s[\mathtt{a}][\mathtt{b}]!\mathtt{m}\langle\mathtt{true}\rangle.X \mid \mu X.s[\mathtt{b}][\mathtt{a}]?\mathtt{m}(x).X$ is deadlock-free, but has a three-party lock.

Proposition 2.1 (3-Plocks). *If P has a 3-plock, then P is not live.*

3 Types, Subtyping and Typing System for MCMπ

This section first introduces types (Def. 3.1), subtyping (Def. 3.2) and properties of typing contexts (Def. 3.4), together with the *three-party lock* for typing contexts (Def. 3.5). We then define the typing system (Def. 3.6) and prove its type soundness (Thm. 3.2).

Syntax and Subtyping. A *session type* is an abstraction of the communications that occur over a channel. A *typing context* is a partial map from channels to closed session types, which mirrors the behaviour of processes.

Definition 3.1 (Types and Typing Contexts). The syntax of *basic types* ($B, B', \ldots$), *session types* ($T, T', \ldots$) and *types* ($S, S', \ldots$) are defined as:

$$B ::= \mathsf{bool} \mid \mathsf{int} \qquad S ::= B \mid T \qquad T ::= \Sigma_{i\in I}\{\mathtt{p}_i\dagger_i\mathtt{m}_i(S_i).T_i\} \mid \mathbf{end} \mid \mu\mathbf{t}.T \mid \mathbf{t}$$

where $I \neq \emptyset$, $\dagger_i \in \{!, ?\}$ and $\{\mathtt{p}_i\dagger_i\mathtt{m}_i\}_{i\in I}$ are distinct, and payload types are closed, and all recursive types are guarded. fv is the free variable function, $\mathsf{roles}(T)$ is the set of participants in T and unfolding is defined by $\mathsf{unf}(\mu\mathbf{t}.T) = T\{^{\mu\mathbf{t}.T}/_{\mathbf{t}}\}$ and $\mathsf{unf}(T) = T$ otherwise. *Typing contexts* are partial functions from channels to closed session types, and their syntax is defined as: $\Delta ::= \emptyset \mid \Delta, c : T$. For

context Δ, Δ_s is the restriction to a session s, $\Delta \setminus s$ is the context with channels over s removed, and $\Delta \setminus c$ is the context with channel c removed. For $T = \Sigma_{i\in I}\{\mathtt{p}_i\dagger_i\mathtt{m}_i(S_i).T_i\}$, write $\mathtt{p}_k\dagger_k\mathtt{m}_k(S_k).T_k \prec T$, $\mathtt{p}_k\dagger_k\mathtt{m}_k(S_k) \prec T$, $\mathtt{p}_k\dagger_k\mathtt{m}_k \prec T$, $\mathtt{p}_k\dagger_k \prec T$ for $k \in I$, and $\mathtt{p}\dagger\mathtt{m}(S)T' \not\prec T$ if $\neg(\mathtt{p}\dagger\mathtt{m}(S)T' \prec T)$. $|T|/|\Delta|$ is the number of constructors in T/Δ. *We assume that all session types are well-formed.*

Definition 3.2 (Mixed Choice Subtyping). The subtyping relation $\leqslant$ between closed types is coinductively defined as:

$$\frac{}{B \leqslant B}\ [\mathrm{B}] \qquad \frac{}{\mathbf{end} \leqslant \mathbf{end}}\ [\mathrm{SEND}] \qquad \frac{|K| > 1 \quad \forall k \in K \quad T_k \leqslant T'_k}{\Sigma_{k\in K}T_k \leqslant \Sigma_{k\in K}T'_k}\ [\mathrm{S}\Sigma]$$

$$\frac{\forall i \in I \quad T_i \leqslant T'_i \quad S'_i \leqslant S_i}{\Sigma_{i\in I}\{\mathtt{p}!\mathtt{m}_i(S_i).T_i\} \leqslant \Sigma_{i\in I\cup J}\{\mathtt{p}!\mathtt{m}_i(S'_i).T'_i\}}\ [\mathrm{SSI}] \qquad \frac{T_1\{\mu\mathbf{t}.T_1/\mathbf{t}\} \leqslant T_2}{\mu\mathbf{t}.T_1 \leqslant T_2}\ [\mathrm{S}\mu\mathrm{L}]$$

$$\frac{\forall i \in I \quad T_i \leqslant T'_i \quad S_i \leqslant S'_i}{\Sigma_{i\in I\cup J}\{\mathtt{p}?\mathtt{m}_i(S_i).T_i\} \leqslant \Sigma_{i\in I}\{\mathtt{p}?\mathtt{m}_i(S'_i).T'_i\}}\ [\mathrm{SBR}] \qquad \frac{T_1 \leqslant T_2\{\mu\mathbf{t}.T_2/\mathbf{t}\}}{T_1 \leqslant \mu\mathbf{t}.T_2}\ [\mathrm{S}\mu\mathrm{R}]$$

We write $\Delta \leqslant \Delta'$ iff $\mathsf{dom}(\Delta) = \mathsf{dom}(\Delta')$ and $\forall c \in \mathsf{dom}(\Delta)$, $\Delta(c) \leqslant \Delta'(c)$; and $\mathsf{end}(\Delta)$ if $\forall c \in \mathsf{dom}(\Delta)$, $\Delta(c) \leqslant \mathbf{end}$; and $\Delta \setminus \mathbf{end}$ for $\{c : T \in \Delta \colon T \not\leqslant \mathbf{end}\}$.

Subtyping $T \leqslant T'$ means that a process implementing T is "less demanding of its channel" than one implementing T' [23,48,18]. Rule [SΣ] from [48] states that subtyping is componentwise; the side condition $|K| > 1$ prevents degenerate derivations. Other rules are standard: in [SSI], fewer internal choices is less demanding; in [SBR], more external choices is less demanding; and in [SEND], terminated types are subtypes of themselves. [SμL] and [SμR] are the standard recursion rules. $\leqslant$ is a preorder on types and typing contexts.

Semantics and Properties. The LTS semantics of session types given below defines behaviours of typed channels. Then the LTS for typing contexts represent behaviours of typed processes, and will be used for typing processes.

Definition 3.3 (The LTS semantics). Let us define *actions* (α,α_i,...) as: $\alpha ::= \mathtt{p}\dagger\mathtt{m}(S) \mid s[\mathtt{p}][\mathtt{q}]{:}\mathtt{m}(S)$. The labelled transition relations on session types and typing contexts are defined as:

$$\frac{k \in I}{\Sigma_{i\in I}\{\mathtt{p}_i\dagger_i\mathtt{m}_i(S_i).T_i\} \xrightarrow{\mathtt{p}_k\dagger_k\mathtt{m}_k(S_k)} T_k}\ [\mathrm{L}\Sigma] \qquad \frac{T\{\mu\mathbf{t}.T/\mathbf{t}\} \xrightarrow{\alpha} T'}{\mu\mathbf{t}.T \xrightarrow{\alpha} T'}\ [\mathrm{L}\mu]$$

$$\frac{T_\mathtt{p} \xrightarrow{\mathtt{q}!\mathtt{m}(S)} T'_\mathtt{p} \quad T_\mathtt{q} \xrightarrow{\mathtt{p}?\mathtt{m}(S')} T'_\mathtt{q} \quad S' \leqslant S}{\Delta, s[\mathtt{p}] : T_\mathtt{p}, s[\mathtt{q}] : T_\mathtt{q} \xrightarrow{s[\mathtt{p}][\mathtt{q}]:\mathtt{m}(S)} \Delta, \mathtt{p} : T'_\mathtt{p}, \mathtt{q} : T'_\mathtt{q}}\ [\mathrm{LCNT}]$$

$\mathtt{p}!\mathtt{m}(S)$ denotes sending with label $\mathtt{m}$ and payload of type S to $\mathtt{p}$; the label $\mathtt{p}?\mathtt{m}(S)$ is its dual; and $s[\mathtt{p}][\mathtt{q}]{:}\mathtt{m}(S)$ a communication from $\mathtt{p}$ to $\mathtt{q}$ using session s with label $\mathtt{m}$ and payload typed by S. A *subject of an action* $\mathsf{subj}(\alpha)$ is defined as: $\mathsf{subj}(\mathtt{p}\dagger\mathtt{m}(S)) = \{\mathtt{p}\}$ and $\mathsf{subj}(s[\mathtt{p}][\mathtt{q}]{:}\mathtt{m}(S)) = \{s[\mathtt{p}], s[\mathtt{q}]\}$. Rule [LΣ] chooses one branch, rule [Lμ] is for a recursion; and rule [LΣ] is for a communication between

dual actions. $\Delta \xrightarrow{\alpha}$ (resp. $T \xrightarrow{\alpha}$) denotes $\exists \Delta'.\Delta \xrightarrow{\alpha} \Delta'$ (resp. $\exists T'.T \xrightarrow{\alpha} T'$). We often write $\rightarrow$ by omitting an action. $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$.

The correctness properties in Def. 2.3 correspond to typing context properties, which are parameters to our typing system.

Definition 3.4 (Properties). The following are common correctness properties: for typing contexts:

- **Safety.** We say that Δ is *safe*, written $\mathsf{safe}(\Delta)$, iff for all $\Delta \rightarrow^* \Delta'$, $\Delta'(s[\mathtt{p}]) \xrightarrow{\mathtt{q}!\mathtt{m}(S)}$ and $\Delta'(s[\mathtt{q}]) \xrightarrow{\mathtt{p}?\mathtt{m}'(S')}$, implies $\Delta' \xrightarrow{s[\mathtt{p}][\mathtt{q}]:\mathtt{m}(S)}$.
- **Deadlock-freedom.** We say that Δ is *deadlock-free*, written $\mathsf{df}(\Delta)$, iff for all $\Delta \rightarrow^* \Delta'$, $\mathsf{end}(\Delta')$ or $\Delta' \rightarrow$.
- **Liveness.** We say that a path $\{\Delta_n\}_{n \in N}$ is:
  - *fair* iff for all $n \in N$, if $\Delta_n \xrightarrow{s[\mathtt{p}][\mathtt{q}]:\mathtt{m}(S)}$, then there exist k, α such that $n \leq k \in N$, $\{s[\mathtt{p}], s[\mathtt{q}]\} \cap \mathsf{sbj}(\alpha) \neq \emptyset$, and $\Delta_k \xrightarrow{\alpha} \Delta_{k+1}$;
  - *live* iff for all $n \in N$, if $\Delta_n(\mathtt{p}) \rightarrow$, then there exist k, α s.t. $n \leq k \in N$, $s[\mathtt{p}] \in \mathsf{sbj}(\alpha)$ and $\Delta_k \xrightarrow{\alpha}$.

  We say that Δ is *live*, written $\mathsf{live}(\Delta)$, iff every fair path starting at Δ is live.

We say that a property φ of typing contexts is *reduction closed* (rc) if $\varphi(\emptyset)$, and $\varphi(\Delta)$ and $\Delta \rightarrow \Delta'$ implies $\varphi(\Delta')$. If φ is rc and $\varphi \subseteq \mathsf{safe}$, then we say that φ is rc-safe e.g. safe, df, and live are rc, and safe, $\mathsf{df} \cap \mathsf{safe}$, and $\mathsf{live} \cap \mathsf{safe}$ are rc-safe.

These properties mirror those for processes (Def. 2.3). We say safe typing contexts are *error-free*. We say a typing context Δ has: a *label mismatch* if $\Delta(s[\mathtt{p}]) \xrightarrow{\mathtt{q}!\mathtt{m}(S)}$, $\Delta(s[\mathtt{q}]) \xrightarrow{\mathtt{p}?\mathtt{m}'(S')}$, and for all S'', $\Delta(s[\mathtt{q}]) \not\xrightarrow{\mathtt{p}?\mathtt{m}(S'')}$; and a *payload mismatch* if $\Delta(s[\mathtt{p}]) \xrightarrow{\mathtt{q}!\mathtt{m}(S)}$, $\Delta(s[\mathtt{q}]) \xrightarrow{\mathtt{p}?\mathtt{m}(S')}$, and $S' \not\leqslant S$.

A context is *safe* if no errors ever occur; it is *deadlock-free* if the only reachable contexts with no further transitions are **end**-valued; and it is *live* if, assuming fair scheduling, all participants that want to communicate will be a subject of some potential communication. A *fair scheduler* ensures that if $\mathtt{p}$ and $\mathtt{q}$ can communicate along s then eventually either (1) $\mathtt{p}$ and $\mathtt{q}$ is scheduled to communicate along s, or (2) one of them communicates with another participant $\mathtt{r}$ along s, disabling a communication. See Ex. 3.2 for why the fair scheduler is needed.

These properties are decidable. See [52] and § 6.1.

Lemma 3.1 (Downwards Closed). *For $\varphi \in \{\mathsf{safe}, \mathsf{safe} \cap \mathsf{df}, \mathsf{safe} \cap \mathsf{live}\}$, if $\Delta' \leqslant \Delta$ and $\varphi(\Delta)$, then $\varphi(\Delta')$.*

Example 3.1 (Simplified Sum Subtyping Rule is Incorrect). If we use the following [SΣBad], as discussed in [48], it violates the subject reduction theorem:

$$\frac{\forall i \in I \quad T_i \leqslant T_i' \quad \text{if } \dagger_i = ?, \text{ then } S_i \leqslant S_i' \text{ else } S_i' \leqslant S_i}{\Sigma_{i \in I}\{\mathtt{p}_i \dagger_i \mathtt{m}_i(S_i).T_i\} \leqslant \Sigma_{i \in I \cup J}\{\mathtt{p}_i \dagger_i \mathtt{m}_i(S_i').T_i'\}} \text{ [SΣBad]}$$

Consider the types: $T_\mathtt{p} = (\mathtt{q}?\mathtt{m} + \mathtt{q}?\mathtt{m}')$, $T_\mathtt{p}' = \mathtt{q}?\mathtt{m}'$ and $T_\mathtt{q} = \mathtt{p}!\mathtt{m}$ and contexts: $\Delta = \{s[\mathtt{p}] : T_\mathtt{p}, s[\mathtt{q}] : T_\mathtt{q}\}$ and $\Delta' = \{s[\mathtt{p}] : T_\mathtt{p}', s[\mathtt{q}] : T_\mathtt{q}\}$. Using [SΣBad], $T_\mathtt{p}'$ would be a subtype of $T_\mathtt{p}$. But Δ is safe while Δ' is not. Hence, unsafe processes typed by Δ' become typable with Δ after applying the subsumption rule ([Sub]).

Example 3.2 (Weak Liveness is not Downwards Closed). Fair paths are necessary in Def. 3.4. Say that Δ is weak live if $\forall \Delta \to^* \Delta'$ and $\forall c \in \mathsf{dom}(\Delta)$, if $\Delta'(c) \to$, then $\Delta' \to^* \Delta'' \xrightarrow{\alpha}$ with $c \in \mathsf{sbj}(\alpha)$. Weak liveness is not downwards closed. Consider: $T'_{\mathtt{p}} = \mu \mathbf{t}.\mathtt{q}!\mathtt{m}.\mathbf{t} \leqslant T_{\mathtt{p}} = \mu \mathbf{t}.\Sigma\{\mathtt{q}!\mathtt{m}.\mathbf{t},\ \mathtt{q}!\mathtt{m}'.\mathtt{r}!\mathtt{m}\}$, $T_{\mathtt{q}} = \mu \mathbf{t}.\Sigma\{\mathtt{p}?\mathtt{m}.\mathbf{t},\ \mathtt{p}?\mathtt{m}'\}$, and $T_{\mathtt{r}} = \mathtt{p}?\mathtt{m}$. The context $\Delta = \{s[\mathtt{p}] : T_{\mathtt{p}}, s[\mathtt{q}] : T_{\mathtt{q}}, s[\mathtt{r}] : T_{\mathtt{r}}\}$ is weak live, but the context $\Delta' = \{s[\mathtt{p}] : T'_{\mathtt{p}}, s[\mathtt{q}] : T_{\mathtt{q}}, s[\mathtt{r}] : T_{\mathtt{r}}\} \leqslant \Delta$ is not. Downwards closedness is important for typing (with [Sub]) to guarantee process liveness (Ex. 5.3).

Example 3.3 (Leader Election with Delegation). We give a typing context for the leader election process in Ex. 2.1: $\Delta_{\text{lead}} = \cup_{0 \leq i \leq 4} \{s_i[\mathtt{p}] : T, s_i[\mathtt{s}] : T', s[\mathtt{p}_i] : T_i\}$ where for $0 \leq i \leq 4$, $T = \mathtt{s}!\mathtt{token}(\mathsf{int})$, and

$$T' = \Sigma\left\{\mathtt{p}_{(i+4)\%5}!\mathtt{elect}(T),\ \mathtt{p}_{(i+3)\%5}!\mathtt{elect}(T),\ \mathtt{p}_{(i+1)\%5}?\mathtt{elect}(T).T'_i\right\},$$
$$T_i = \Sigma\left\{\mathtt{p}_{(i+4)\%5}!\mathtt{elect}(T),\ \mathtt{p}_{(i+3)\%5}!\mathtt{elect}(T),\ \mathtt{p}_{(i+1)\%5}?\mathtt{elect}(T).T'_i\right\}$$
$$T'_i = \Sigma\left\{\mathtt{p}_{(i+2)\%5}!\mathtt{elect}(T),\ \mathtt{p}_{(i+3)\%5}?\mathtt{elect}(T).\mathtt{p}_{(i+2)\%5}?\mathtt{elect}(T)\right\}$$

Observe that $T \leqslant T'' = \Sigma\{\mathtt{s}!\mathtt{token}(\mathsf{int}), \mathtt{s}!\mathtt{token}'(\mathsf{int})\}$, thus

$$T'_0 \not\leqslant \Sigma\{\mathtt{p}_2!\mathtt{elect}(T),\ \mathtt{p}_3?\mathtt{elect}(T'').\mathtt{p}_2?\mathtt{elect}(T'')\} \leqslant T'_0$$
$$T'_0 \leqslant \Sigma\{\mathtt{p}_2!\mathtt{elect}(T''),\ \mathtt{p}_3?\mathtt{elect}(T).\mathtt{p}_2?\mathtt{elect}(T)\}$$

As with the calculus, Δ_{lead} is safe, deadlock-free, and live.

Three-Party Locks in typing contexts are defined below.

Definition 3.5 (Three-Party Lock (3-plock)). Δ has a *three-party lock (3-plock)* iff $\mathsf{unf}(\Delta(s[\mathtt{p}])) = \Sigma_{i \in I}\{\mathtt{q}_i \dagger_i \mathtt{m}_i(S_i).T_i\}$, and for all $i \in I$, if $s[\mathtt{q}_i] \in \mathsf{dom}(\Delta)$, either: (1) $\mathsf{unf}(\Delta(s[\mathtt{q}_i])) = \mathbf{end}$; or (2) $\mathsf{unf}(\Delta(s[\mathtt{q}_i])) = \Sigma_{j \in J_i}\{\mathtt{r}_i \dagger'_i \mathtt{m}_{i,j}(S_{i,j}).T_{i,j}\}$ and if $s[\mathtt{r}_i] \in \mathsf{dom}(\Delta)$ then either: (a) $\mathtt{r}_i = \mathtt{p}$ and $\dagger'_i = \dagger_i$; or (b) $\mathsf{unf}(\Delta(s[\mathtt{r}_i])) = \mathbf{end}$; or (c) $\mathsf{unf}(\Delta(s[\mathtt{r}_i])) = \Sigma_{j \in J'_i}\{\mathtt{p} \dagger''_i \mathtt{m}'_{i,j}(S'_{i,j}).T'_{i,j}\}$. The context Δ is 3-plock free iff for all $\Delta \to^* \Delta'$, Δ' has no 3-plock.

These mirror the conditions in Def. 2.4. $\mathsf{top}(P, s[\mathtt{p}][\mathtt{q}]\dagger)$ says that the only action that $\mathtt{p}$ can take on channel s, if there are any, is $\dagger$ towards $\mathtt{q}$. The equivalent for types is $s[\mathtt{p}] \notin \mathsf{dom}(\Delta)$, or $\mathsf{unf}(\Delta(s[\mathtt{p}])) = \mathbf{end}$, or $\mathsf{unf}(\Delta(s[\mathtt{p}])) = \Sigma_{i \in I}\{\mathtt{q} \dagger \mathtt{m}_i(S_i).T_i\}$. Def. 3.5 corresponds to Def. 2.4 with $\mathsf{top}(P, s[\mathtt{p}][\mathtt{q}]\dagger)$ replaced by its equivalent for types.

Example 3.4 (Three-Party Lock). We provide typing contexts for Ex. 2.2:

$$\Delta_1 = \{s[\mathtt{p}] : \mathtt{q}!\mathtt{m}(\mathsf{int}), s[\mathtt{q}] : \mathbf{end}\} \quad \Delta_2 = \{s[\mathtt{p}] : \mathtt{q}!\mathtt{m}(\mathsf{int}), s[\mathtt{q}] : \mathtt{p}!\mathtt{m}(\mathsf{int})\}$$
$$\Delta_3 = \{s[\mathtt{p}] : \mathtt{q}!\mathtt{m}(\mathsf{int}), s[\mathtt{q}] : \mathtt{r}?\mathtt{m}(\mathsf{int}), s[\mathtt{r}] : \mathtt{p}!\mathtt{m}(\mathsf{int})\}$$

All three have 3-plocks. Δ_1 has a 3-plock by Def. 3.5(1) since $\Delta_1(s[\mathtt{q}]) = \mathbf{end}$. Participant $\mathtt{p}$ cannot communicate along s since it can only communicate with $\mathtt{q}$, which has no further behaviour. Similarly, Δ_2 and Δ_3 have 3-plocks by Def. 3.5(2-a) and (2-c), respectively.

Typing System and its Type Soundness. We define the typing system and prove subject reduction and error-freedom.

$$\frac{\Gamma, X{:}\Delta \vdash P : \Delta \quad \mathsf{dom}(\Delta \setminus \mathbf{end}) \subseteq \mathsf{chan}(P, \Gamma)}{\Gamma \vdash \mu X.P : \Delta}\ [\textsc{Rec}] \qquad \frac{}{\Gamma, X{:}\Delta \vdash X : \Delta}\ [\textsc{Var}] \qquad \frac{}{\Gamma \vdash \mathbf{0} : \emptyset}\ [\textsc{Nil}]$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash P : \Delta \quad \Gamma \vdash P' : \Delta}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P' : \Delta}\ [\textsc{If}] \qquad \frac{\Gamma \vdash P : \Delta \quad \Gamma \vdash P' : \Delta'}{\Gamma \vdash P \mid P' : \Delta, \Delta'}\ [\textsc{Par}]$$

$$\frac{\Gamma \vdash P : \Delta \quad \varphi(\Delta_s)}{\Gamma \vdash (\nu s)\, P : \Delta \setminus s}\ [\textsc{Res}] \qquad \frac{\Gamma \vdash P : \Delta \quad \Delta \leqslant \Delta' \quad \mathsf{end}(\Delta'')}{\Gamma \vdash P : \Delta', \Delta''}\ [\textsc{Sub}]$$

$$\frac{c \neq c' \quad \mathtt{q}!\mathtt{m}(\Delta(c')).T'' \prec \Delta(c) \quad \Gamma \vdash P : (\Delta \setminus c \setminus c'), c : T''}{\Gamma \Vdash c[\mathtt{q}]!\mathtt{m}\langle c' \rangle.P : \Delta}\ [\textsc{!Chan}] \qquad \frac{\mathtt{q}!\mathtt{m}(B).T'' \prec \Delta(c) \quad \Gamma \vdash e : B \quad \Gamma \vdash P : (\Delta \setminus c), c : T''}{\Gamma \Vdash c[\mathtt{q}]!\mathtt{m}\langle e \rangle.P : \Delta}\ [\textsc{!Val}]$$

$$\frac{\mathtt{q}?\mathtt{m}(T').T'' \prec \Delta(c) \quad \Gamma \vdash P : (\Delta \setminus c), c : T'', y : T'}{\Gamma \Vdash c[\mathtt{q}]?\mathtt{m}(y).P : \Delta}\ [\textsc{?Chan}] \qquad \frac{\mathtt{q}?\mathtt{m}(B).T'' \prec \Delta(c) \quad \Gamma, x{:}B \vdash P : (\Delta \setminus c), c : T''}{\Gamma \Vdash c[\mathtt{q}]?\mathtt{m}(x).P : \Delta}\ [\textsc{?Val}]$$

$$\frac{\forall j \in I \left(\begin{array}{c} \Gamma \Vdash \pi_j.P_j : \Delta \wedge (c = \mathsf{subj}(\pi_j) \implies \\ \forall \mathtt{q}\dagger\mathtt{m}(S).T \prec \Delta(c).\ c[\mathtt{q}]\dagger\mathtt{m} \prec \sum_{i \in I} \pi_i.P_i \end{array} \right)}{\Gamma \vdash \sum_{i \in I} \pi_i.P_i : \Delta}\ [\textsc{Sum}]$$

Fig. 2: Typing Rules. Parameterised by typing context property φ.

Definition 3.6 (Type Judgements). Let Γ denote a *typing environment* (Γ, Γ',...) defined by $\Gamma ::= \emptyset \mid \Gamma, X : \Delta \mid \Gamma, x : B$. Judgement $\Gamma \vdash e : B$ is standard for typing expressions [23, Table 4]. Let φ in [Res] in Fig. 2 be rc-safe. We define $\mathsf{chan}(P, \Gamma) = \mathsf{chan}(P) \cup \bigcup_{X \in \mathsf{fpv}(P)} \mathsf{dom}(\Gamma(X))$, the channels appearing in P with reference to an environment. We define the process typing judgement, written $\Gamma \vdash P : \Delta$, inductively by the rules in Fig. 2.

The new typing rules (highlighted in yellow) are [Rec], [Sum], [!Chan], [!Val], [?Chan], and [?Val]; [Rec] handles recursion, and the others handle mixed choice. [Rec] types recursive processes with a context. [Sum] consists of two parts: (1) if the channel c occurs in a sum, then each choice in $\Delta(c)$ must occur in the sum; (2) each choice over c in the sum must occur in $\Delta(c)$, which is checked by the judgement $\Gamma \Vdash \pi.P : \Delta$. The judgement $\Gamma \Vdash \pi.P : \Delta$ is defined by the rules [!Chan], [!Val], [?Chan], and [?Val]. [!Chan] and [!Val] handle sending channels and values, respectively; and [?Chan] and [?Val] dually handle receiving. Note that in the session delegation ([!Chan]), after c delegates session endpoint c' to its receiver, P can no longer hold c' ($\Delta \setminus c \setminus c'$) since c' must be used linearly.

Example 3.5 (Typing Leader Election). Recall P_{lead} from Ex. 2.1 and Δ_{lead} from Ex. 3.3. Derive $\emptyset \vdash P_{\text{lead}} : \Delta_{\text{lead}}$ by: [Nil] at $\mathbf{0}$ then [Sub] to introduce **end** typed channels; [Sum] at each sum subterm; and [Par] at each composition subterm.

Our typing system satisfies subject reduction, yielding error-freedom.

Theorem 3.1 (Subject Reduction). *Let φ in [Res] in Fig. 2 be rc-safe. Suppose that $\Gamma \vdash P : \Delta$ and $\mathsf{safe}(\Delta)$. 1. If $P \Rrightarrow P'$, then $\Gamma \vdash P' : \Delta$. 2. $P \to P'$, then either $\Gamma \vdash P' : \Delta$ or there exists Δ' such that $\Delta \to \Delta'$ and $\Gamma \vdash P' : \Delta'$.*

Corollary 3.1 (Error-freedom). *Let φ in* [Res] *in Fig. 2 be rc-safe. If $\Gamma \vdash P : \Delta$ with $\mathsf{safe}(\Delta)$ and $P \to^* P'$, then P' contains no errors.*

Theorem 3.2 (Correctness). *Let φ in* [Res] *in Fig. 2 be rc-safe. If $\Gamma \vdash P : \Delta$ with $\mathsf{safe}(\Delta)$, then P is safe. If $\varphi \subseteq \mathsf{live}$, then P is 3-plock free.*

Recall that $\mathsf{safe} \cap \mathsf{live}$ is rc-safe (Def. 3.4) and $\mathsf{safe} \cap \mathsf{live} \subseteq \mathsf{live}$. Using $\mathsf{safe} \cap \mathsf{live}$ in [Res] in Fig. 2, typable processes are 3-plock free. This does *not* guarantee liveness of typed process P, since P might contain interleaved sessions. See § 5.

Example 3.6 (Interleaving). Let $P = s[\mathtt{q}][\mathtt{p}]?\mathtt{m}.s'[\mathtt{p}][\mathtt{q}]!\mathtt{m} \mid s'[\mathtt{q}][\mathtt{p}]?\mathtt{m}.s[\mathtt{p}][\mathtt{q}]!\mathtt{m}$ and $\Delta = s[\mathtt{p}] : \mathtt{q}!\mathtt{m}, s[\mathtt{q}] : \mathtt{p}?\mathtt{m}, s'[\mathtt{p}] : \mathtt{q}!\mathtt{m}, s'[\mathtt{q}] : \mathtt{p}?\mathtt{m}$. We derive the judgement: $\vdash P : \Delta$. Although $\mathsf{safe}(\Delta) \cap \mathsf{df}(\Delta) \cap \mathsf{live}(\Delta)$, P is neither deadlock-free nor live.

Example 3.7 (Leader Election with Delegation). From Ex. 3.5 and Thm. 3.2, P_{lead} is safe and is 3-plock free.

4 Operational and Denotational Preciseness

This section defines and proves soundness and completeness (hence preciseness) of subtyping both operationally (Thm. 4.6) and denotationally (Thm. 4.7).

4.1 Operational Preciseness

We first define *operational preciseness* following [40,13,23,24]: a subtyping is *sound* if all its instances are sound; a subtyping is *complete* if it contains *all* sound instances; and a subtyping is *precise* if it is sound and complete.

Definition 4.1 (Operational Preciseness). Let $\varphi = \mathsf{live} \cap \mathsf{safe}$ in [Res] in Fig. 2. We say that a subtyping instance (T', T) is *sound* iff for all roles $\mathtt{p} \notin \mathsf{roles}(T)$, sessions s, contexts $\mathbb{C}[]$, processes P, and typing contexts Δ:

$$\begin{array}{ll} \text{if} & (\forall Q.\ \emptyset \vdash Q : \{s[\mathtt{p}] : T\} \implies \emptyset \vdash \mathbb{C}[Q] : \emptyset) \\ \text{then} & (\emptyset \vdash P : \{s[\mathtt{p}] : T'\} \implies \mathbb{C}[P] \text{ is safe and 3-plock free.}) \end{array}$$

Let $\trianglelefteq$ be a preorder on closed session types. We say that $\trianglelefteq$ is *sound* if $T' \trianglelefteq T$ implies that (T', T) is a sound subtyping instance; $\trianglelefteq$ is *complete* if for all sound subtyping instances (T', T), $T' \trianglelefteq T$; and $\trianglelefteq$ is *precise* if it is sound and complete.

Theorem 4.1 (Operational Soundness). *The subtyping $\leqslant$ is sound.*

Proof. Suppose that $T' \leqslant T$ and for all roles $\mathtt{p} \notin \mathsf{roles}(T)$, sessions s, contexts $\mathbb{C}[]$, processes Q, and typing contexts Δ, if $\emptyset \vdash Q : \{s[\mathtt{p}] : T\}$ then $\emptyset \vdash \mathbb{C}[Q] : \emptyset$.

By [Sub], if $\emptyset \vdash P : \{s[\mathtt{p}] : T'\}$ then $\emptyset \vdash P : \{s[\mathtt{p}] : T\}$. By assumption, $\emptyset \vdash \mathbb{C}[P] : \emptyset$. Thus by Thm. 3.2, $\mathbb{C}[P]$ is safe and 3-plock free. □

We prove completeness in four main steps.

[Step 1]: We construct live contexts for every type, T, which check for every disallowed behaviour. The detailed construction is described in Def. 4.2. The contexts without T are called *complementary contexts*.

[Step 2]: Using **Step 1**, we construct *characteristic processes* for every type T and channel c (Def. 4.4).

[Step 3]: We show that the negation of subtyping has an inductive definition (Thm. 4.4).

[Step 4]: By induction on **Step 3**, the constructions in **Step 2** and **Step 1** provide contexts witnessing completeness.

Complementary Contexts. Step 1 has a subtle point since we must be able to check for every disallowed behaviour. However, types can only encode finitely many choices. Therefore, we must limit the tested types with a known and finite set of labels, payloads and roles.

Let $\mathcal{T}$ be a finite set of session types such that for all $T \in \mathcal{T}$, if T' appears in T as a payload then $T' \in \mathcal{T}$. Let $L/\mathcal{R}$ be the set of labels/participants appearing within $\mathcal{T}$. We say that a type T is $\mathcal{T}$-*valid* iff all payload session types in T are in $\mathcal{T}$, all labels in T are in L, and all participants in T are in $\mathcal{R}$. We use $\mathcal{T}$-valid types to handle all possible mixed choices that may appear.

Definition 4.2 (Complementary Types). Fix a finite set of session types, $\mathcal{T}$. We use $\mathcal{T}$ to enumerate the possible messages, $\mathtt{m}(S)$, and construct a collection of message sets, Act.

We define $\mathsf{wait}(\mathtt{q})$ to be the type of $\mathtt{q}$ in a complementary context; and $\mathsf{sch}(T)$ for $\mathcal{T}$-valid types T to be the type of the scheduler in a complementary context for T. We write $\mathcal{A}[s, T] = s[\mathtt{sch}] : \mathsf{sch}(T), \bigcup_{\mathtt{q}\in\mathcal{R}} s[\mathtt{q}] : \mathsf{wait}(\mathtt{q})$, the complementary context for T and s, for $\mathcal{T}$-valid T. The full definition is given below:

Fix a finite set of session types $\mathcal{T}$ such that for all $T \in \mathcal{T}$, if T' is a payload type in T then $T' \in \mathcal{T}$. Let L be the set of labels appearing within $\mathcal{T}$. Let $\mathcal{R}$ be the set of participants appearing within $\mathcal{T}$. Enumerate:

$$\mathcal{R} = \{\mathtt{r}_i\}_{0\le i\le n} \qquad \{\mathtt{m}(S) : \mathtt{m} \in L, S \in \mathcal{T} \cup \{\mathsf{int}, \mathsf{bool}\}\} = \{\mathtt{m}_i(S_i)\}_{0\le i\le N-1}$$

Let $Act = \{I \subseteq \{0, \dots, N-1\} : \forall i \neq j \in I.\ \mathtt{m}_i \neq \mathtt{m}_j\}$, the set of all message sets, where messages have distinct labels. Let $\mathtt{p}$ and $\mathtt{sch}$ be fresh roles, and $\mathtt{m}''$ be a fresh label i.e. $\mathtt{p}, \mathtt{sch} \notin \mathcal{R}$ and $\mathtt{m}'' \notin L$. We first define type $\mathsf{wait}(\mathtt{q})$:

$$\mathsf{wait}(\mathtt{q}) = \mu\mathbf{t}.\big(\Sigma_{I\in Act,\dagger\in\{!,?\}}\{\mathtt{sch}?\mathtt{m}_{(I,\dagger)}.\mathsf{pfm}(\mathtt{q}, I, \dagger, \mathbf{t})\} + \Sigma_{\dagger\in\{!,?\}}\{\mathtt{sch}?\mathtt{m}'_\dagger.\mathsf{det}(\dagger, \mathbf{t})\} + \Sigma_{\mathtt{r}\in\mathcal{R}\setminus\{\mathtt{q}\}}\{\mathtt{sch}?\mathtt{m}_i.\mathsf{sync}?(\mathtt{r}_i, \mathbf{t})\} + \mathtt{sch}?\mathtt{m}_{\mathrm{end}}\big)$$

with $\mathsf{sync}?(\mathtt{q}, T) = \mathtt{q}?\mathtt{m}.T$ $\quad \mathsf{sync}!(\mathtt{r}_k, T) = \mathtt{r}_0!\mathtt{m}.\dots.\mathtt{r}_{k-1}!\mathtt{m}.\mathtt{r}_{k+1}!\mathtt{m}.\dots.\mathtt{r}_n!\mathtt{m}.T$

$\mathsf{det}(\dagger, T) = \mathtt{sch}!\mathtt{m}.T + \mathtt{p}\dagger\mathtt{m}''$ $\quad \mathsf{pfm}(\mathtt{q}, I, \dagger, T) = \Sigma_{i\in I}\{\mathtt{p}\dagger\mathtt{m}_i(S_i).\mathsf{sync}!(\mathtt{q}, \mathtt{sch}!\mathtt{m}_i.T)\}$

Next we define $\mathsf{sch}(T)$ for $\mathcal{T}$-valid types T, by the following recursive definition:

$$\mathsf{sch}(\mathbf{end}) = \mathtt{r}_0!\mathtt{m}_{\mathrm{end}} \dots \mathtt{r}_n!\mathtt{m}_{\mathrm{end}} \qquad \mathsf{sch}(\mathbf{t}) = \mathbf{t} \qquad \mathsf{sch}(\mu\mathbf{t}.T) = \mu\mathbf{t}.\mathsf{sch}(T)$$

$$\mathsf{sch}\Big(\Sigma_{(\mathtt{q},\dagger)\in J,\ i\in I_{(\mathtt{q},\dagger)}}\{\mathtt{q}\dagger\mathtt{m}_i(S_i).T_{i,\mathtt{q},\dagger}\}\Big) = \sum_{(\mathtt{q},\dagger)\in\mathcal{R}\times\{!,?\}\setminus J}\mathsf{test}\Big(\mathtt{q}, \overline{\dagger}, \sum_{(\mathtt{q},\dagger)\in J}\mathsf{trigger}\big(\mathtt{q}, I_{(\mathtt{q},\dagger)}, \overline{\dagger}, \{\mathsf{sch}(T_{i,\mathtt{q},\dagger})\}_{i\in I_{(\mathtt{q},\dagger)}}\big)\Big)$$

with $\mathsf{trigger}(\mathtt{q}, I, \dagger, \{T_i\}_{i\in I}) = \mathtt{q}!\mathtt{m}_{(I,\dagger)}.\mathsf{sync}(\mathtt{q}, \Sigma_{i\in I}\{\mathtt{q}?\mathtt{m}_i.T_i\})$ $\quad \mathsf{test}(\mathtt{q}, \dagger, T) = \mathtt{q}!\mathtt{m}'_\dagger.\mathtt{q}?\mathtt{m}.T$

$\mathsf{sync}(\mathtt{r}_k, T) = \mathtt{r}_0!\mathtt{m}_k \dots \mathtt{r}_{k-1}!\mathtt{m}_k.\mathtt{r}_{k+1}!\mathtt{m}_k \dots \mathtt{r}_n!\mathtt{m}_k.T$

- The information in $\mathcal{T}$ is used to restrict the possible payloads, labels, and roles, that the complementary context must account for.
- Our live context for T is $\Delta[s, T], s[\mathtt{p}] : T$.
- For all $\mathtt{q} \in \mathcal{R}$, $\mathsf{wait}(\mathtt{q})$ responds to commands from $\mathtt{sch}$ to either: terminate; check if $\mathtt{p}$ erroneously contains the action $\mathtt{q}\overline{\dagger}$; communicate with $\mathtt{p}$ with $\dagger$ and the message set $\{\mathtt{m_i}(S_i)\}_{i\in I}$; or synchronise with another participant.
- In $\mathsf{wait}(\mathtt{q})$, $\mathsf{sync!}(\mathtt{q}, T)$ sends a message from $\mathtt{q}$ to the other participants; $\mathsf{sync?}(\mathtt{q}, T)$ is a dual receiver; $\mathsf{pfm}(\mathtt{q}, I, \dagger, T)$ allows $\mathtt{q}$ to communicate with $\mathtt{p}$ with action $\dagger$, and $\mathsf{det}(\dagger, T)$ allows $\mathtt{q}$ to test for the action of $\mathtt{q}\overline{\dagger}$ in $\mathtt{p}$.
- $\mathsf{sch}(T)$ is the scheduling type. It tells other participants how to communicate with $\mathtt{p}$ to check erroneous actions and to make any all acceptable actions.
- In $\mathsf{sch}(T)$, $\mathsf{sync}(\mathtt{q}, T)$ lets $\mathtt{sch}$ command all $\mathtt{p} \in \mathcal{R} \setminus \{\mathtt{q}\}$ to receive a synchronisation from $\mathtt{q}$; $\mathsf{test}(\mathtt{q}, \dagger, T)$ lets $\mathtt{sch}$ command $\mathtt{q}$ to test $\mathtt{q}\overline{\dagger}$ at $\mathtt{p}$; and $\mathsf{trigger}(\mathtt{q}, I, \dagger, \{T_i\}_{i\in I})$ lets $\mathtt{sch}$ command $\mathtt{q}$ to communicate with $\mathtt{p}$ with $\dagger$.

Example 4.1 (Complementary Types). Consider the types $T = \mathtt{q}!\mathtt{m}(\mathbf{end})$ and $T' = \mathtt{r}!\mathtt{m}(\mathbf{end})$. Let $\mathcal{T} = \{T, T', \mathbf{end}\}$ so that T and T' are $\mathcal{T}$-valid, $\mathcal{R} = \{\mathtt{q}, \mathtt{r}\}$, and $Act = \{\{\mathtt{m}(\mathbf{end})\}, \{\mathtt{m}(\mathsf{int})\}, \{\mathtt{m}(\mathsf{bool})\}\}$. We have that:

$$\mathsf{sch}(T) = \Sigma\{\mathtt{q}!\mathtt{m}'_!.\mathtt{q}?\mathtt{m}.T'',\ \mathtt{r}!\mathtt{m}'_!.\mathtt{r}?\mathtt{m}.T'',\ \mathtt{r}!\mathtt{m}'_?.\mathtt{r}?\mathtt{m}.T''\}$$
$$T'' = \mathtt{q}!\mathtt{m}_{(\{\mathtt{m}(\mathbf{end})\},?)}.\mathtt{r}!\mathtt{m}_{\mathtt{q}}.\mathtt{q}?\mathtt{m}.\mathtt{q}!\mathtt{m}_{\mathrm{end}}.\mathtt{r}!\mathtt{m}_{\mathrm{end}}$$

The scheduler will first choose an action not present in T i.e. not a prefix of $\mathsf{unf}(T)$ ($\mathtt{q}!$, $\mathtt{r}!$, or $\mathtt{r}?$), and tell its participant to test for it. If $\mathtt{p}$ implements the chosen action, then a label mismatch error occurs. Otherwise, the participant returns back to the scheduler, which continues with T''. The scheduler tells $\mathtt{q}$ to receive the message set $\{\mathtt{m}(\mathbf{end})\}$ from $\mathtt{p}$, and tells $\mathtt{r}$ to synchronise with $\mathtt{q}$. Once $\mathtt{q}$ receives a message from $\mathtt{p}$ and synchronises with $\mathtt{r}$, $\mathtt{q}$ informs the scheduler of the outcome of the communication. The scheduler now believes that $\mathtt{p}$ should have the type $\mathbf{end}$, so tells $\mathtt{q}$ and $\mathtt{r}$ to terminate.

Note that this differentiates T' and T, since T' contains the action $\mathtt{r}!$. The scheduler may tell $\mathtt{r}$ to check for $\mathtt{r}!$, inducing a label mismatch.

Characteristics Processes. For **Step 2**, we introduce characteristic processes for types and contexts. For this purpose, we introduce the following helper values and processes to check the types of values. Below v_B is a value of type B and $P_B(x)$ is a process that is unsafe iff x is not of type B and terminates otherwise.

Definition 4.3 (Basic Type Characteristics). *Let* $v_{\mathsf{bool}} = \mathit{true}$, $v_{\mathsf{int}} = 0$, $P_{\mathsf{int}}(x) = \mathsf{if}\ \mathit{succ}(x)\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathbf{0}$, $P_{\mathsf{bool}}(x) = \mathsf{if}\ x\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathbf{0}$.

The characteristic process of a session type T, as defined below in Def. 4.4, is a process that: 1. behaves as specified by T; and 2. checks that received values and channels behave as expected in T.

Definition 4.4 (Characteristic Processes). *For channel c and type T, with $\mathcal{T}$-valid payload types, we define the process $\mathcal{P}[c, T]$ by recursion on T.*

For a context Δ, we write $\mathcal{P}[\Delta] = \Pi_{c\in\Delta}\mathcal{P}[c, \Delta(c)]$.

$$\mathcal{P}[c, \mathbf{end}] = \mathbf{0} \qquad \mathcal{P}[c, \mu \mathbf{t}.T] = \mu X_{\mathbf{t}}.\mathcal{P}[c, T] \qquad \mathcal{P}[c, \mathbf{t}] = X_{\mathbf{t}}$$

$$\mathcal{P}[c, \Sigma_{i\in I}\{\mathtt{q}_\mathtt{i}\dagger_i \mathtt{m}_\mathtt{i}(S_i).T_i\}] = (\nu s_i \colon i \in I')\left(\textstyle\sum_{i\in I} \pi_i.P_i \mid \Pi_{i\in I''}\mathcal{P}[\mathcal{A}[s_i, S_i]]\right)$$

where $I' = \{i \in I \colon \dagger_i = !, S_i \notin \{\mathsf{int}, \mathsf{bool}\}\}$, $I'' = \{i \in I' \colon S_i \neq \mathbf{end}\}$

$$\pi_i.P_i = \begin{cases} c[\mathtt{q}_i]!\mathtt{m}_\mathtt{i}\langle s_i[\mathtt{p}]\rangle.\left(\mathcal{P}[c, T_i] \mid \mathcal{P}\left[\bigcup_{j\in I'\setminus\{i\}} s_j[\mathtt{p}] : S_j\right]\right) & \dagger_i = !, S_i \notin \{\mathsf{int}, \mathsf{bool}\} \\ c[\mathtt{q}_i]!\mathtt{m}_\mathtt{i}\langle v_{S_i}\rangle.\left(\mathcal{P}[c, T_i] \mid \mathcal{P}\left[\bigcup_{j\in I'} s_j[\mathtt{p}] : S_j\right]\right) & \dagger_i = !, S_i \in \{\mathsf{int}, \mathsf{bool}\} \\ c[\mathtt{q}_i]?\mathtt{m}_\mathtt{i}(y).\left(\mathcal{P}[c, T_i] \mid \mathcal{P}[y, S_i] \mid \mathcal{P}\left[\bigcup_{j\in I'} s_j[\mathtt{p}] : S_j\right]\right) & \dagger_i = ?, S_i \notin \{\mathsf{int}, \mathsf{bool}\} \\ c[\mathtt{q}_i]?\mathtt{m}_\mathtt{i}(x).\left(\mathcal{P}[c, T_i] \mid P_{S_i}(x) \mid \mathcal{P}\left[\bigcup_{j\in I'} s_j[\mathtt{p}] : S_j\right]\right) & \dagger_i = ?, S_i \in \{\mathsf{int}, \mathsf{bool}\} \end{cases}$$

Characteristic processes are designed to mimic semantics of a type and a complementary context whenever a session endpoint is created for using a payload.

Characteristic process $\mathcal{P}[c, T]$ follows the structure of T along channel c. To output values of type B, the characteristic value v_B is used. To output channels of type T', a new session, s', is created with channel $s'[\mathtt{p}]$ having type T' as enforced by the introduction of characteristic process of a complementary context of T'. Received values are verified at runtime by the characteristic processes of basic types. Received channels are verified at runtime by characteristic process of their expected types. Leftover channel endpoints, created for sends that did not occur, are used by characteristic processes of their expected types.

Proposition 4.1 (Characteristic Processes). *For channel c and type T, with $\mathcal{T}$-valid payload types, the recursive definition for $\mathcal{P}[c, T]$ is terminates.*

Proposition 4.2 (Complementary Context). *If T is $\mathcal{T}$-valid and closed, then* $(\mathsf{safe} \cap \mathsf{live})(\mathcal{A}[s, T], s[\mathtt{p}] : T)$.

A session type can be used to type its characteristic process.

Theorem 4.2 (Characteristic Processes). *Let T be $\mathcal{T}$-valid. Write* $\mathsf{fv}(T) = \{\mathbf{t}_1, \ldots \mathbf{t}_n\}$ *and let $T_1, \ldots, T_n$ be closed and $\mathcal{T}$-valid.*

$$\{X_{\mathbf{t}_i} : \{c : T_i\}\}_{i\leq n} \vdash \mathcal{P}[c, T] : \{c : T\{{}^{T_1}/_{\mathbf{t}_1}\} \ldots \{{}^{T_n}/_{\mathbf{t}_n}\}\}$$

Specifically, $\emptyset \vdash \mathcal{P}[c, T] : \{c : T\}$ for closed and $\mathcal{T}$-valid T. By subject reduction, this says that T captures the behaviour of its characteristic process $\mathcal{P}[c, T]$. The reverse is also true: Lem. 4.1 states that $\mathcal{P}[c, T]$ captures behaviours of T.

Example 4.2 (Characteristic Process). Consider the type:

$$T = \Sigma\{\mathtt{q}!\mathtt{m}'(\mathbf{end}),\ \mathtt{q}!\mathtt{m}(\mathsf{int}).\mathtt{q}?\mathtt{m}(\mathtt{r}!\mathtt{m}(\mathsf{bool}))\}$$

We construct its characteristic process by performing the specified behaviour and handling the received channel:

$$\mathcal{P}[s[\mathtt{p}], T] = (\nu s')\left(s[\mathtt{p}][\mathtt{q}]!\mathtt{m}'\langle s'[\mathtt{p}]\rangle + s[\mathtt{p}][\mathtt{q}]!\mathtt{m}'\langle 0\rangle.s[\mathtt{p}][\mathtt{q}]?\mathtt{m}(y).y[\mathtt{r}]!\mathtt{m}\langle \mathtt{true}\rangle\right)$$

Observe that: $\emptyset \vdash y[\mathtt{r}]!\mathtt{m}\langle \mathtt{true}\rangle : \{y : \mathtt{r}!\mathtt{m}(\mathsf{bool})\}$ $\emptyset \vdash \mathcal{P}[s[\mathtt{p}], T] : \{s[\mathtt{p}] : T\}$.
We use [Nil] at $\mathbf{0}$, [Sub] to introduce **end**-valued channels, [Sum] at all the sum processes, and [Res] at $(\nu s')$ since $\mathsf{live}(\{s'[\mathtt{p}] : \mathbf{end}\})$.

$$\frac{T'\{^{\mu \mathbf{t}.T'}/_{\mathbf{t}}\} \not\lhd T}{\mu \mathbf{t}.T' \not\lhd T}\ [\mu\text{L}] \qquad \frac{\mathtt{q}!\mathtt{m} \prec T' \quad \mathtt{q}!\mathtt{m} \not\prec T}{T' \not\lhd T}\ [!\text{L}] \qquad \frac{\mathtt{q}? \prec T' \quad \mathtt{q}? \not\prec T}{T' \not\lhd T}\ [?\text{L}]$$

$$\frac{T' \not\lhd T\{^{\mu \mathbf{t}.T}/_{\mathbf{t}}\}}{T' \not\lhd \mu \mathbf{t}.T}\ [\mu\text{R}] \qquad \frac{\mathtt{q}?\mathtt{m} \prec T \quad \mathtt{q}?\mathtt{m} \not\prec T'}{T' \not\lhd T}\ [?\text{R}] \qquad \frac{\mathtt{q}! \prec T \quad \mathtt{q}! \not\prec T'}{T' \not\lhd T}\ [!\text{R}]$$

$$\frac{\mathtt{q}?\mathtt{m}(S') \prec T' \quad \mathtt{q}?\mathtt{m}(S) \prec T \quad S' \not\lhd S}{T' \not\lhd T}\ [\text{LR?}] \qquad \frac{}{\mathbf{end} \not\lhd \Sigma_{i \in I} T_i}\ [\mathbf{end}\text{-L}] \qquad \frac{S \neq B}{B \not\lhd S}\ [B\text{-L}]$$

$$\frac{\mathtt{q}!\mathtt{m}(S') \prec T' \quad \mathtt{q}!\mathtt{m}(S) \prec T \quad S \not\lhd S'}{T' \not\lhd T}\ [\text{LR!}] \qquad \frac{}{\Sigma_{i \in I} T_i \not\lhd \mathbf{end}}\ [\mathbf{end}\text{-R}] \qquad \frac{S \neq B}{S \not\lhd B}\ [B\text{-R}]$$

$$\frac{\mathtt{q}\dagger\mathtt{m}(S').T''' \prec T' \quad \mathtt{q}\dagger\mathtt{m}(S).T'' \prec T \quad T''' \not\lhd T''}{T' \not\lhd T}\ [\text{LR}]$$

Fig. 3: Negation of Subtyping Rules

Theorem 4.3 (Two-Party Locks are Insufficient). *There are types $T' \not\leqslant T$ such that for all Δ, if $\Delta, s[\mathtt{p}] : T$ is live, then $\Delta, s[\mathtt{p}] : T'$ is 2-plock free.*

Proof. Δ has a two-participant lock (2-plock) iff $\mathsf{unf}(\Delta(s[\mathtt{p}])) = \Sigma_{i \in I}\{\mathtt{q}_\mathtt{i} \dagger_i \mathtt{m}_\mathtt{i}(S_i).T_i\}$ and, for all $i \in I$ if $s[\mathtt{q}_i] \in \mathsf{dom}(\Delta)$ then, $\mathsf{unf}(\Delta(s[\mathtt{q}_i])) = \mathbf{end}$ or $\mathsf{unf}(\Delta(s[\mathtt{q}_i])) = \Sigma_{j \in J_i}\{\mathtt{p} \dagger \mathtt{m}_{\mathtt{i},\mathtt{j}}(S_{i,j}).T_{i,j}\}$.

Let $T = (\mathtt{q}?\mathtt{m}.\mathtt{r}?\mathtt{m}' + \mathtt{r}?\mathtt{m}.\mathtt{q}?\mathtt{m}')$ and $T' = \mathtt{q}?\mathtt{m}.\mathtt{r}?\mathtt{m}'$. Suppose that $\Delta, s[\mathtt{p}] : T$ is live. If $\Delta \rightarrow^* \Delta'$ and $\Delta', s[\mathtt{p}] : T' \xrightarrow{s[\mathtt{q}][\mathtt{p}]:\mathtt{m}} \Delta''$, then $\Delta, s[\mathtt{p}] : T \rightarrow^* \Delta''$, so Δ'' is live and is 2-plock free. $\Delta', s[\mathtt{p}] : T'$ has no two-participant locks: $\Delta'(s[\mathtt{q}]) \rightarrow^* T''$ without $\mathtt{p}$ s.t. $T'' \xrightarrow{\mathtt{p}!\mathtt{m}''}$, by following a fair transition sequence of $\Delta', s[\mathtt{p}] : T$ until $s[\mathtt{p}]$ terminates; so $\mathsf{unf}(\Delta'(s[\mathtt{q}])) \neq \mathbf{end}$ and $\mathsf{unf}(\Delta'(s[\mathtt{q}])) \neq \Sigma_{i \in I}\{\mathtt{p}?\mathtt{m}_\mathtt{i}(S_i)\}$. □

Note that the proof of Thm. 3.2 shows that if a context has no 2-plocks, then neither does any process that it types. Thus, Thm. 4.3 also proves that 2-plocks for processes are insufficient for preciseness.

Negation of Subtyping. As for **Step 3**, recall that subtyping is defined coinductively (Def. 3.1). The negation of coinductive relations are inductive, suggesting that we can find inductive rules for $\not\leqslant$. Indeed, Thm. 4.4 proves this.

Theorem 4.4 (Negation of Subtyping). *Define the binary relation $T' \not\lhd T$ on closed session types inductively by the rules in Fig. 3, then $T' \not\lhd T$ iff $T' \not\leqslant T$.*

Example 4.3 (Negation of Subtyping). Consider the types $T = \mathtt{q}!\mathtt{m}(\mathbf{end})$ and $T' = \mathtt{r}!\mathtt{m}(\mathbf{end})$. $T' \not\lhd T$ by: [!L] as $\mathtt{r}!\mathtt{m} \prec T'$ and $\mathtt{r}!\mathtt{m} \not\prec T$; or by [!R] as $\mathtt{q}! \prec T$ and $\mathtt{q}! \not\prec T'$. Thus, $\mathtt{q}?\mathtt{m}'(T') \not\lhd \mathtt{q}?\mathtt{m}'(T)$ by [LR?].

A similar theorem is crucial in [23,13] where preciseness was proved by induction on 'failing derivations' of subtyping.

Operational Completeness. To proceed with **Step 4**, we observe that the semantics of characteristic processes derive from the semantics of their types

(Lem. 4.1). Also observe that complementary contexts fully explore their types, in the sense that every transition of T is used within the some path from $\mathcal{A}[s, T], s[\mathtt{p}] : T$. From this, we deduce operational completeness.

Lemma 4.1 (Characteristic Fidelity). *(1) If $\Delta \rightarrow \Delta'$, then $\mathcal{P}[\Delta] \rightarrow^* Q \mid \mathcal{P}[\Delta']$; (2) If $\mathtt{q}!\mathtt{m}(T) \prec \Delta(s[\mathtt{r}])$ and $\mathtt{r}?\mathtt{m}(T') \prec \Delta(s[\mathtt{q}])$, then $\mathcal{P}[\Delta] \rightarrow^* Q \mid (\nu s')\, \mathcal{P}[\mathcal{A}[s', T], s'[\mathtt{p}] : T']$ if $T \neq \mathbf{end}$ and $\mathcal{P}[\Delta] \rightarrow^* Q \mid (\nu s')\, \mathcal{P}[s'[\mathtt{p}], T']$ if $T = \mathbf{end}$; (3) If Δ has a label or payload mismatch, with one being a basic type, then $\mathcal{P}[\Delta] \rightarrow^* \mathbb{C}[\mathtt{err}]$; and (4) If Δ has a 3-plock on s, then so does $(\nu s)\, \mathcal{P}[\Delta]$.*

The behaviours of a type T (up to subtyping) are precisely those making $\mathcal{A}[s, T]$ safe and live i.e. if T' does not behave according to T (is not a subtype), then $\mathcal{A}[s, T]$ can detect this, causing an error in the characteristic process.

Theorem 4.5 (Characteristic Strictness). *If $T' \not\leqslant T$ are $\mathcal{T}$-valid and closed, then there is P such that $(\nu s)\, (\mathcal{P}[\mathcal{A}[s, T]] \mid \mathcal{P}[s[\mathtt{p}], T']) \rightarrow^* P$ and P has an error or a 3-plock.*

Proof. Induction on the derivation of $T' \not\trianglelefteq T$, using Lem. 4.1. □

Theorem 4.6 (Preciseness). *The subtyping $\leqslant$ is operationally precise.*

Proof. Let $T' \not\leqslant T$ and fix $\mathcal{T}$ making T' and T $\mathcal{T}$-valid. If $\emptyset \vdash Q : \{s[\mathtt{p}] : T\}$, then $\emptyset \vdash \mathcal{P}[\mathcal{A}[s, T]] \mid Q : (\mathcal{A}[s, T], s[\mathtt{p}] : T)$, hence $\emptyset \vdash (\nu s)\, (\mathcal{P}[\mathcal{A}[s, T]] \mid Q) : \emptyset$. $\emptyset \vdash \mathcal{P}[s[\mathtt{p}], T'] : \{s[\mathtt{p}] : T'\}$ and $(\nu s)\, (\mathcal{P}[\mathcal{A}[s, T]] \mid \mathcal{P}[s[\mathtt{p}], T'])$ is not safe or is not 3-plock free by Thm. 4.5. Therefore, (T', T) is not a sound subtyping instance. Therefore, $\leqslant$ is complete. By Thm. 4.1, $\leqslant$ is sound, so $\leqslant$ is precise. □

4.2 Denotational Preciseness

We define the interpretation of closed type T to be the set of processes which T types: $[\![T]\!] = \{P \colon \emptyset \vdash P : \{y : T\}\}$. We say that a preorder $\trianglelefteq$ is **denotationally precise** if for all T' and T, $T' \trianglelefteq T$ iff $[\![T']\!] \subseteq [\![T]\!]$.

Theorem 4.7 (Denotational Preciseness). *The subtyping $\leqslant$ is denotationally precise.*

Proof. If $T' \leqslant T$, then $[\![T']\!] \subseteq [\![T]\!]$ by [Sub]. If $[\![T']\!] \subseteq [\![T]\!]$, then fix $\mathcal{T}$ with T' and T $\mathcal{T}$-valid. By [Sub], $\mathcal{P}[y, T'] \in [\![T]\!]$, so $\emptyset \vdash (\nu s)\, (\mathcal{P}[\mathcal{A}[s, T]] \mid \mathcal{P}[s[\mathtt{p}], T']) : \emptyset$, thus $(\nu s)\, (\mathcal{P}[\mathcal{A}[s, T]] \mid \mathcal{P}[s[\mathtt{p}], T'])$ is safe and 3-plock free. By Thm. 4.5, $T' \leqslant T$. □

5 Extensions to Mixed Choice Multiparty Subcalculi

We define a typing system (Def. 5.1) for processes without session interleaving and delegation (Prop. 5.1) and show that type checking guarantees liveness and deadlock-freedom (Thm. 5.3). Using this typing system, we define preciseness with respect to liveness (Def. 5.2) and show that the subtyping, $\leqslant$, is precise with

respect to liveness (Thm. 5.4). We conclude this section with further extensions of our preciseness result (Def. 5.4) to four multiparty session subcalculi of mixed choice given in [48] (Def. 5.3).

Session Fidelity, Deadlock-Freedom and Liveness As in the previous work, deadlock-freedom and liveness of typed processes are not generally guaranteed. We limit the calculus to one without session delegation and interleaving by using the following restricted type judgement.

Definition 5.1 (Restricted Type Judgement). *We define the typing judgement, $\Gamma \vdash^\star P : \Delta$, inductively by the rules in Fig. 2, replacing* [Sum] *by:*

$$\frac{\forall j \in I\ \forall \mathtt{q}\dagger\mathtt{m}(S).T' \prec T\,(\mathsf{end}(\Delta) \wedge \Gamma \Vdash \pi_j.P_j : (\Delta, c:T) \wedge c[\mathtt{q}]\dagger\mathtt{m} \prec \textstyle\sum_{i\in I}\pi_i.P_i)}{\Gamma \vdash^\star \sum_{i\in I}\pi_i.P_i : (\Delta, c:T)}\ [\textsc{Sum}^*]$$

[Sum*] differs from [Sum] since it disallows non-deterministic choice over distinct channels and the only channel whose reducts are hidden by a communication involving c is c itself. Therefore, we minimally remove session interleaving and non-trivial session delegation. This is less restrictive than the 'plays role' restriction in [52], and retains all the same guarantees.

Proposition 5.1 (Session Delegation). *If $\Gamma \vdash^\star P : \Delta$, then all instances of session delegation in the derivation are trivial i.e. if $\Gamma' \vdash^\star P' : \Delta'$ is a subderivation and $c'[\mathtt{p}]!\mathtt{m}\langle c\rangle \prec P'$, then* $\mathbf{end} \leqslant \Delta'(c)$.

Example 5.1 (Unbounded Leader Election wihout Delegation). Let $\varphi = \mathsf{safe}\cap\mathsf{live}$ in [Res] in Fig. 2. Recall P_{lead} from Ex. 2.1. This process contains non-trivial session delegation (sent channels are used once received), thus it is not typable with the restricted type judgement (Prop. 5.1). We remove delegation to reattain typability. This restricted typed calculus is still interesting, as it can type a revised leader election with unbounded session creations. Consider an extension, where the elected participant spawns a new session running the protocol.

$$Q_{\text{lead}} = \mu X.(\nu s)\,(Q_0 \mid Q_1 \mid Q_2 \mid Q_3 \mid Q_4) \quad \text{where, for } 0 \le i \le 4$$

$$Q_i = s[\mathtt{p}_i][\mathtt{p}_{(i+4)\%5}]!\mathtt{elect}\langle \mathtt{i}\rangle + s[\mathtt{p}_i][\mathtt{p}_{(i+1)\%5}]?\mathtt{elect}(x).Q_i' + s[\mathtt{p}_i][\mathtt{p}_{(i+3)\%5}]!\mathtt{elect}\langle \mathtt{i}\rangle$$
$$Q_i' = s[\mathtt{p}_i][\mathtt{p}_{(i+2)\%5}]!\mathtt{elect}\langle \mathtt{i}\rangle + s[\mathtt{p}_i][\mathtt{p}_{(i+3)\%5}]?\mathtt{elect}(x').s[\mathtt{p}_i][\mathtt{p}_{(i+2)\%5}]?\mathtt{elect}(x'').X$$

We derive $\emptyset \vdash^\star Q_{\text{lead}} : \emptyset$.

Theorem 5.1 (Subject Reduction). *Let φ in* [Res] *in Fig. 2 be rc-safe. Suppose that $\Gamma \vdash^\star P : \Delta$ and $\mathsf{safe}(\Delta)$. 1. If $P \Rrightarrow P'$, then $\Gamma \vdash^\star P' : \Delta$. 2. $P \to P'$, then either $\Gamma \vdash^\star P' : \Delta$ or there exists Δ' s.t. $\Delta \longrightarrow \Delta'$ and $\Gamma \vdash^\star P' : \Delta'$.*

Theorem 5.2 (Session Fidelity). *Let φ in* [Res] *in Fig. 2 be rc-safe. Suppose that P is guarded, $\mathsf{safe}(\Delta)$, and $\emptyset \vdash^\star P : \Delta$. If $\Delta \longrightarrow$, then there exist P' and Δ' such that $\Delta \longrightarrow \Delta'$, $P \to^* P'$, and $\emptyset \vdash^\star P' : \Delta'$.*

Theorem 5.3 (Properties). *Let φ in* [Res] *in Fig. 2 be rc-safe. Suppose that P is guarded, $\emptyset \vdash^\star P : \Delta$, and $\varphi(\Delta)$: (1) If $\varphi \subseteq \mathsf{df}$, then P is deadlock-free; (2) If $\varphi \subseteq \mathsf{live}$, then P is live.*

Example 5.2 (Leader Election without Delegation). From Ex. 5.1 and Thm. 5.3, we deduce that the process Q_{lead} is live.

Example 5.3 (Fairness is Necessary for Liveness). Due to [Sub] in Fig. 2, the fact that $\mathsf{safe} \cap \mathsf{live}$ is downwards closed (Lem. 3.1) is crucial. Safe and weak live (liveness without fairness) is not downwards closed (Ex. 3.2), so it can type processes that are not live. Recall $\Delta' \leqslant \Delta$ from Ex. 3.2, where Δ is weak live, Δ' is not, $\mathsf{safe}(\Delta)$, and $\mathsf{safe}(\Delta')$. Consider the process

$$P = \mu X.(\nu s')\, s[\mathtt{p}][\mathtt{q}]!\mathtt{m}\langle s'[\mathtt{p}]\rangle.X \mid \mu X.\sum \begin{pmatrix} s[\mathtt{q}][\mathtt{p}]?\mathtt{m}(y).X \\ s[\mathtt{q}][\mathtt{p}]?\mathtt{m}'(y).\mathbf{0} \end{pmatrix} \mid s[\mathtt{r}][\mathtt{p}]?\mathtt{m}'(y).\mathbf{0}$$

P is not live; the reduct $s[\mathtt{r}][\mathtt{p}]?\mathtt{m}'(y).\mathbf{0}$ can never be used. However, $\emptyset \vdash^\star P : \Delta'$ so by [Sub], $\emptyset \vdash^\star P : \Delta$ (independent of φ in [Res] in Fig. 2).

Preciseness of a Familiy of Mixed Choice π-Calculi. Without session delegation, the subtyping $\leqslant$ is precise w.r.t. liveness.

Definition 5.2 (Operational Preciseness w.r.t. Liveness). We say that a subtyping instance (T', T) is *sound w.r.t. liveness* similarly to Def. 4.1 but with: $\varphi = \mathsf{live} \cap \mathsf{safe} \cap \{\Delta\colon \Delta \text{ contains no session payloads}\}$; $\Gamma \vdash^\star P : T''$; T and T' only having basic typed payloads; and requiring process liveness.

Let $\trianglelefteq$ be a preorder on closed session types. We say that $\trianglelefteq$ is: *sound w.r.t. liveness* if ($T' \trianglelefteq T$ and T' and T only have basic typed payloads) implies that (T', T) is a sound w.r.t. liveness subtyping instance; *complete w.r.t. liveness* if for all sound w.r.t. liveness subtyping instances (T', T), $T' \trianglelefteq T$; and *precise w.r.t. liveness* if it is both sound and complete w.r.t. liveness.

Remark 5.1 (Precise w.r.t. Liveness). The terminology 'precise w.r.t. liveness' is taken from [24]. It is used as in Def. 5.2, where the typing system is sound only if typed processes are live. Although safety/error-freedom is not mentioned, it is still required for soundness.

Theorem 5.4 (Preciseness w.r.t. Liveness). *The subtyping $\leqslant$ is operationally and denotationally precise w.r.t. liveness.*

Proof. Thm. 5.3 implies soundness. Replace the **end** valued payloads in Def. 4.2 by int and remove all $I \in \mathit{Act}$ s.t. $\exists i \in I$ s.t. S_i is a session type. Following the proof in § 4.1, we derive completeness since 3-plocks are livelocks. □

We now extend our result to the family of multiparty subcalculi in [48]:

Definition 5.3 (Subcalculi of Mixed Choice).

1. Mixed Separate Choice per Participant (*MS*). Each participant in a sum type cannot be both send to and received from i.e. in $\Sigma_{i\in I}\{\mathtt{q}_i \dagger_i \mathtt{m}_i(S_i).T_i\}$, $\forall i, j \in I.\mathtt{q}_i = \mathtt{q}_j \implies \dagger_i = \dagger_j$.

2. Separate Choice (SC). In a sum type, either every choice is a send, or every choice is a receive i.e. the sum syntax is given by $\Sigma_{i\in I}\{\mathtt{q}_i\dagger\mathtt{m}_i(S_i).T_i\}$.
3. Directed Mixed Choice (DM). All choices in a sum type are with the same participant i.e. the sum syntax is given by $\Sigma_{i\in I}\{\mathtt{q}\dagger_i\mathtt{m}_i(S_i).T_i\}$.
4. Separated Choice (S). Sum types are either a sum of outputs to one participant, or a sum of inputs from one participant i.e. $\Sigma_{i\in I}\{\mathtt{q}\dagger\mathtt{m}_i(S_i).T_i\}$.

Note that these subcalculi allow session delegations unless specified.

For each $\mathbb{F}\in\{MS, SC, DM, S\}$, the subtyping $\leqslant$, restricted to types in $\mathbb{F}$, is precise w.r.t. $\mathbb{F}$. If we further restrict $\leqslant$ to types without session-typed payloads, the subtyping $\leqslant$ is precise w.r.t. $\mathbb{F}$ and liveness.

Definition 5.4 (Operational Preciseness with respect to Subcalculi). Let $\mathbb{F}\in\{MS, SC, DM, S\}$ be a subcalculus of mixed choice. We say that a subtyping instance (T', T) is *sound* w.r.t. $\mathbb{F}$ similarly to Def. 4.1 but with: $\varphi = \mathsf{live}\cap\mathsf{safe}\cap\{\Delta\colon \Delta \text{ is in } \mathbb{F}\}$; and T and T' are the type syntax defined in $\mathbb{F}$. Let $\trianglelefteq$ be a preorder on session types. We say that $\trianglelefteq$ is: *sound w.r.t.* $\mathbb{F}$ if ($T'\trianglelefteq T$ and T, T' are in $\mathbb{F}$) implies that (T', T) is a sound w.r.t. $\mathbb{F}$ subtyping instance; *complete w.r.t.* $\mathbb{F}$ if for all sound w.r.t. $\mathbb{F}$ subtyping instances (T', T), $T'\trianglelefteq T$; and *precise w.r.t.* $\mathbb{F}$ if it is both sound and complete w.r.t. $\mathbb{F}$.

Theorem 5.5 (Preciseness w.r.t. Subsystems). *For $\mathbb{F}\in\{MS, SC, DM, S\}$, the subtyping, $\leqslant$, is operationally precise and denotationally w.r.t. $\mathbb{F}$ and operationally and denotationally precise w.r.t. $\mathbb{F}$ and liveness.*

For MS, it is immediate since complementary contexts (Def. 4.2) are MS. For $\mathbb{F}\in\{MS, SC, DM, S\}$, remove the disallowed syntax from Def. 4.2 and proceed as in § 4.1.

6 Implementations of Subtyping and Properties

This section first gives the algorithms for checking properties and their complexity results, and then explains our tool implementation and benchmark results.

6.1 Deciding Context Properties

For a context Δ, let: $\mathsf{barb}(\Delta) = \mathsf{dom}(\Delta\setminus\mathbf{end})$, the set of barbs; $\mathsf{obs}(\Delta) = \{\mathsf{sbj}(\alpha)\colon \Delta\xrightarrow{\alpha}\}$, the set of observations; and $\mathsf{R}(\Delta, \Delta', c)$ be true iff there is a path from Δ to Δ', where c is never observed i.e. $\mathsf{R}(\Delta, \Delta', c)$ iff there exists a path $\{\Delta_i\}_{0\leq i\leq n}$ such that $\Delta = \Delta_0$, $\Delta' = \Delta_n$, and for all $i \leq n$, $c\notin\bigcup\mathsf{obs}(\Delta_i)$. Similarly to [55, Lem. 6.17], we can rewrite fair and not live paths as a finite cycle; we can divide this cycle into smaller cycles. From this we derive Prop. 6.1.

Proposition 6.1 (Liveness). *Δ is not live iff there exists Δ' s.t. $c\in\mathsf{barb}(\Delta')$ and $\Delta\rightarrow^*\Delta'$ and for all $X\in\mathsf{obs}(\Delta')$ there exist $\Delta''\xrightarrow{\alpha}\Delta'''$ s.t. $\mathsf{R}(\Delta', \Delta'', c)$, $\mathsf{R}(\Delta''', \Delta', c)$, and $X\cap\mathsf{sbj}(\alpha)\neq\emptyset$.*

Deciding subtyping is necessary to construct the LTS ([LCnt] in Def. 3.3). We apply the subtyping algorithm in [55, Thm. 3.13], extended with mixed choice.

Theorem 6.1 (Deciding Subtyping). *Let T and T' be type graphs ([55, Def. 3.6]). Checking $T' \leqslant T$ has worst-case complexity $\mathcal{O}(|T'| \cdot |T|)$.*

Theorem 6.2 (Deciding Properties). *Let Δ be a typing context with $|\Delta| = n$, $m = |\{\Delta' \colon \Delta \rightarrow^* \Delta'\}|$, and $|\mathsf{dom}(\Delta)| = r$. Safety, deadlock-freedom, and liveness can all be decided in either: (1) time $\mathcal{O}\left(rn^2m^2\right)$ and space $\mathcal{O}\left(n^2 + rm^2\right)$; or (2) time $\mathcal{O}\left(rn^2m^2 \log n\right)$ and space $\mathcal{O}\left(n^2 + rnm\right)$.*

The time complexities are achieved by enumerating the state space as it is generated, recording $\mathsf{obs}(\Delta')$, $\mathsf{barb}(\Delta')$, and the states adjacent to Δ', as Δ' is explored. (1) is achieved by using adjacency arrays; and (2) is by using adjacency sets. Safety and deadlock-freedom are decided by checking each state for label/-payload mismatches and termination while having barbs, respectively. Liveness is decided by checking the LTS against the predicate in Prop. 6.1 directly.

6.2 Implementation and Benchmarks

We implemented the session type and typing context syntax and semantics, from this paper, in Scala 3.82. For typing context semantics, we implemented the subtyping algorithm in Thm. 6.1. Using these, we implemented the liveness predicate from Prop. 6.1 as well as safety and deadlock-freedom. First, we construct an LTS from a typing context; during the construction, we infer whether the context is safe/deadlock-free. Then, we compute the reachability relation. Finally, we loop over all roles, p; for each one we compute the 'reachability without observing p' relation, and loop over all contexts to check if the inner predicate is satisfied. This decides if the predicate is satisfied in time $\mathcal{O}(rn^2m^2)$, where r is the size of the domain, n is the size of the context, and m is the size of the state space. The optimisation of using type graphs, instead of syntactic types that are manually unfolded, is provided, and is used during benchmarking.

Our machine configurations are: Windows 11 Home Version 25H2; AMD Ryzen 7 4700U @ 2.00 GHz; 16 GB RAM. Each benchmark is run multiple times before recording 20 benchmarks, from which we calculate a mean.

Benchmarks evaluating our tool are given in the Table 1. For each example we report: which subcalculi the example is within; the parameter used if applicable; the number of states in the LTS; the mean time to determine safety in ms; the mean time to determine deadlock-freedom in ms; and the mean time to determine liveness in ms.

Our implementation is nearly instant on all handwritten examples. While property checking is not instant on our large parameterised examples, it is still practical with the largest mean being checking liveness of the leader election protocol from [26] with 8 participants. This context's LTS has 18350 states and the property checker terminates in 31 seconds. Larger examples, taking longer, exceed the memory capacity of personal computers for both this work and `mpstk`

Example	*MC*	*MS*	*SC*	*B*	#*N*	#states	safe	df	live
OAuth2 [52]					N/A	5	0.9	0.9	0.9
TwoBuyer [52]					N/A	7	0.1	0.1	0.1
MapReduce [52]					3	20	1.1	1.1	1.25
					5	42	0.75	0.75	1.05
					10	132	5.35	5.35	9.85
					50	2652	5065.15	5065.25	13786.5
MPWorker [52]					3	156	5.65	5.65	11.05
					5	3906	3219.75	3219.85	8005.5
BinaryCounter					3	125	1.85	1.85	3.3
					5	3125	1000.0	1000.05	1991.55
BadBinaryCounter	✓				3	1375	157.65	157.65	186.25
LeaderElection [48]	✓				N/A	22	0.15	0.15	0.3
ServerRequests [34]			✓		2	5	0.05	0.05	0.05
					10	21	0.2	0.2	0.55
					100	201	181.25	181.25	237.75
LoadBalancing [41]			✓		2	4	0.1	0.1	0.1
					10	12	0.1	0.1	0.25
					100	102	57.8	57.8	74.45
Calculator [31]				✓	N/A	5	0.05	0.05	0.05
SMTP [31]				✓	N/A	8	0.05	0.05	0.05
CircuitBreaker [35]			✓		N/A	14	0.15	0.15	0.2
DistributedLogging [35]				✓	N/A	4	0.0	0.0	0.05
TravelAgent [32]				✓	N/A	6	0.0	0.0	0.0
OnlineWallet [44]					N/A	10	0.05	0.05	0.05
LeaderElection [26]		✓			3	40	1.25	1.25	1.35
					5	467	16.45	16.55	27.45
					8	18350	14428.35	14428.6	30586.9
Running [7]			✓		N/A	6	0.0	0.0	0.0
ClientServerWorkers [3]		✓			N/A	11	0.1	0.1	0.1
TimeOut [3]		✓			N/A	6	0.05	0.05	0.05
Fire [3]	✓				N/A	13	0.05	0.05	0.05
LeaderElection (Ex. 3.3)		✓			N/A	17	0.1	0.1	0.2
Cycle (**??**)	✓				2	3	0.05	0.05	0.05
					3	33	0.2	0.2	0.25
					4	125	2.0	2.0	2.85
					5	359	12.9	12.9	14.4
					10	10743	15640.3	15640.45	17455.35

Table 1: Benchmarks for liveness algorithm implementation. *MC* denotes that full mixed choice is necessary; *MS*, mixed separate choice per participant; *SC*, separate choice; *B*, binary (2-party) sessions; and #*N* denotes the parameter for the example, if applicable. Mean times are given in ms.

(a tool used in [52]) before time becomes a practical issue. Benchmarks for `mpstk` are omitted as the non mixed choice benchmarks are available in [52,5,50] and are around 100 times slower than our implementation.

Notice that the average times for checking safe and df are similar, but lower than the times for live. This is because safe and df can be immediately inferred during construction of the LTS, whereas checking live requires constructing the LTS and performing further processing.

7 Related Work and Conclusion

Preciseness of Subtyping. The canonicity of subtyping relations in functional calculi has been assessed by denotational soundness and completeness, defined by interpretations of types. Denotational completeness of the λ-calculus was first considered in [4] for arrow and intersection types, followed by [56,33,58] extending to union and pair types. Operational preciseness was first defined by Ligatti et al. [39] for the call-by-value λ-calculus with iso-recursive types. Following this work, Dezani and Ghilezan [20] show operational preciseness of subtyping for a concurrent λ-calculus with arrow, intersection, and union types.

The first work on preciseness for session types [13] has shown that the synchronous [11] and asynchronous subtyping relations for binary session types with session delegation are precise with respect to safety and 'co-channel duality' in their π-calculus. Their paper [14] summarises the current state of the art on precise subtyping. The technique developed for binary session types is insufficient for multiparty communication (Thm. 4.3): we require 3-plocks to take the place of 'co-channel duality' (2-plocks).

Ghilezan et al. [23,24] proved preciseness with respect to safety and liveness, for synchronous and asynchronous subtypings in multiparty session calculi. The top-down approach for precise asynchronous subtyping is proposed in [49]. The proof strategy in [23,24] follows the steps given in § 1. There are, however, significant technical differences from the proofs in this paper. In Step 1, we use a scheduler that has a collection of possible messages (Def. 4.2) in contrast to [23,24] which use cyclic communications to give participants control over their communications. Cyclic communications do not work in mixed choice as multiple participants may act simultaneously. Related to Step 2, [23,24] leave session delegation and creation and interleaving in characteristic processes as an open problem, which has been closed in this paper (Def. 4.4). In Step 3, we directly characterise the negation of subtyping instead of using an intermediate algorithm as in [23]. In Step 4, the use of a full π-calculus with interleaving required us to introduce 3-plocks (Def. 2.4) instead of simply using errors or referring to processes getting 'stuck'. This allows us to have a more general type system and requires us to handle delegation and manipulation of 3-plocks in our lemmas (Lem. 4.1). Additionally, mixed choice causes the negation of subtyping to have more cases than in [23], adding comprehensive proof cases which do not occur in separated choice. Furthermore, we have discovered that [23]'s operational sound-

ness is flawed because their subject reduction theorem fails for their projection. The full merge coinductive projection from [55] might fix this issue.

Li et al. [37] propose a decidable subtyping relation for communicating automata [9] which is sound and complete with respect to *subprotocol fidelity* and deadlock-freedom. This work considers correctness w.r.t. implementing global types, which is not related to either process correctness or typing systems. Padovani and Zavattaro [46] propose a subtyping for binary session types with asynchronous semantics that is complete with respect to *fair termination*, and Padovani et al. [10] propose one that is complete with respect to a *convergence* relation. Both completeness results are w.r.t. type semantics not their typing system; hence it differs from our preciseness.

Bottom-Up Mixed Choice and Tools. The non-deterministic sum syntax originated with the π-calculus [43], and has since incited work analysing its expressive power. Palamidessi [47] showed that there is no 'good' encoding of the π-calculus with guarded mixed choice into the π-calculus with guarded separate choice. This was followed by [48], giving the expressiveness hierarchy of session calculi with various restrictions of choice. The subtyping in [48] is the first extension of the standard subtyping to mixed choice and, by fixing rule [SΣ], is equal to our subtyping. Prokić et al. [51] have formalised and typed federated learning protocols with asynchronous separated choice. Their subtyping is the restriction of our subtyping to separated choice types, which is precise under synchronous semantics but not asynchronous semantics. Hinrichsen et al. [26] introduced 'Mixtris', a separation logic for multiparty message-passing programs. Mixtris is used to reason about *functional* correctness using dependent binders, and neither formulates nor guarantees either deadlock-freedom or liveness. While their functional correctness is intrinsically undecidable, we have: proved decidability for deadlock-freedom and liveness; implemented an algorithm to decide these properties; and provided its complexity. Blechschmidt et al. [7] use a probabilistic mixed choice multiparty session π-calculus; this additionally has probabilistic selections but excludes session delegation and only allows choice over a single channel. They introduced the 'refinement subtyping', which permits replacement of channels by several channels implementing the same behaviour. This differs from the standard approach of defining a subtyping on session types and extending to typing contexts pointwise.

The tool `mpstk` [52] and its extension [6] decide typing context properties for bottom-up MPST systems. They encode properties as μ-calculus formulae [52, Figure 5] to use mCRL2 for model checking. This has two main limitations: `mpstk` does not support mixed choice; the formula for liveness in [52, Figure 5] is not applicable for mixed choice liveness; and the correctness of the formulae has been left unproven. We instead propose algorithms to check typing context properties, which are more efficient than the approach in [52]. Our algorithm has no external dependencies, thus it is more extendable for a future adaptation.

Typing systems for π-calculi often fail to guarantee deadlock-freedom and liveness due to interleaving and delegation (Ex. 3.6). Work to overcome this is

inapplicable to this paper: [16] relies on global types, [45,57] use binary session types and none of them use mixed choice.

Top-Down Mixed Choice. The first work providing a top-down approach with mixed choice is [19], where they explore a relationship between communicating automata [9] and multiparty session types. Later, Lange et al. [36] studied a synthesis algorithm which builds a global choreography with mixed choice from communicating automata. Castagna et al. [12] provide a semantic description of well-formedness for asynchronous mixed choice global types and a semantic notion of projection with a decidable approximation. Jongmans and Yoshida [34] extended global types with mixed choice, existential quantification, and unrestricted parallel composition. They verify well-formedness by checking bisimilarity between projected contexts and global types. Majumdar et al. [41] and Li et al. [38] provide projection algorithms for sender-driven global types under asynchronous semantics. This is a restriction of mixed choice where each global choice has the same sender. The algorithm in [38] is complete for deadlock-free communicating automata [9]. Hamers and Jongmans [25] implemented mixed choice session types in 'Discourje', for run-time verification of Clojure programs. Cruz-Filipe et al. [17] consider flexible choice for choreographies, but as 'multicoms/multisels' instead of mixed choices.

Realisability of mixed choice global types in asynchronous semantics with respect to deadlock-free communicating automata [9] has been proven undecidable in [53]. Bocchi et al. [8] propose new global syntax and semantics, which allow for the transient inconsistencies that may occur during non-deterministic communications. Di Giusto et al. [22] approach realisability without projecting. They provide algorithms, for both synchronous and asynchronous (p2p) semantics, that decide whether a global type is realisable, given a complementary global type. These cannot be found in general for mixed choice types. Barbanera and Dezani have considered mixed choice global types, synchronously [2] and asynchronously [3]. They introduced notions of coherence between sets of global choices and local contexts to deconstruct sessions into modules. This differs from standard projection algorithms as it relates a global type to an entire context at once, and the algorithm is parameterised by a partition of the participant set. This causes issues with scaling, since components cannot be considered individually and useful partitions are non-trivial to provide. They use a session calculus (similar to local types), which lacks session interleaving and delegation.

None of the above works prove the preciseness of subtyping relations.

Conclusion and Future Work. The preciseness result of this paper shows that the mixed choice multiparty subtyping $\leqslant$ (Def. 3.2) is the canonical notion of subtyping for the full synchronous multiparty session calculus in § 2; it is both operationally precise (Thm. 4.6) and denotationally precise (Thm. 4.7).

Establishing preciseness requires the introduction of new machinery and proof methodologies, together with detailed analysis and technically involved proofs. Two of our main technical contributions are 3-plocks (Def. 2.4) and complementary contexts (Def. 4.2). 3-plocks are minimal patterns of livelocks and is the part of context liveness that is maintained by typing processes with dele-

gation and session interleaving (Thm. 3.2). Complementary context $\mathcal{A}[s, T]$ for each type T, monitors all possible failures of subtyping non-deterministically and forces errors or 3-plocks if any incorrect behaviour is detected. These proof techniques are not required for two-party or separated choice, shedding light on the expressiveness of multiparty mixed choice.

We proposed a new typing system for a mixed choice multiparty session calculus which ensures safety (Thm. 3.2), and deadlock-freedom and liveness (Thm. 5.3). Our third main technical contribution is the representation of live in Prop. 6.1. This substantially improves the complexity of deciding live (Thm. 6.2), making it tractable in the size of the state space and context. The benchmarking of our implementation (§6.2) has shown that this approach is efficient in practice.

Our future work includes: preciseness for asynchronous mixed choice multiparty session types; and the top-down approach with mixed choice, building a limited but practical projection algorithm along the line of [8].

Acknowledgements. We thank the reviewers for their detailed comments. This work was supported by EPSRC EP/T006544/2, EP/T014709/2, EP/Z533749/1, ARIA and Horizon EU TaRDIS 101093006 (UKRI number 10066667).

References

1. Anonymous. Mixed choice multiparty session type property checking, May 2026. doi:10.5281/zenodo.20428250.
2. Franco Barbanera and Mariangiola Dezani-Ciancaglini. Modular multiparty sessions with mixed choice. In Clément Aubert, Cinzia Di Giusto, Simon Fowler, and Violet Ka I Pun, editors, *Proceedings 18th Interaction and Concurrency Experience, ICE 2025, Lille, France, 20th June 2025*, EPTCS, pages 2–20, August 2025. doi:10.4204/EPTCS.425.2.
3. Franco Barbanera and Mariangiola Dezani-Ciancaglini. Asynchronous multiparty sessions with mixed choice. *CoRR*, abs/2604.06872, 2026. URL: https://doi.org/10.48550/arXiv.2604.06872, arXiv:2604.06872, doi:10.48550/ARXIV.2604.06872.
4. Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symb. Log.*, 48(4):931–940, December 1983.
5. Adam Barwell, Alceste Scalas, Nobuko Yoshida, and Fangyi Zhou. Generalised Multiparty Session Types with Crash-Stop Failures. In *33rd International Conference on Concurrency Theory*, volume 243 of *LIPIcs*, pages 35:1–35:25. Dagstuhl, 2022. doi:10.4230/LIPIcs.CONCUR.2022.35.
6. Adam D. Barwell, Ping Hou, Nobuko Yoshida, and Fangyi Zhou. Crash-Stop Failures in Asynchronous Multiparty Session Types. *Logical Methods in Computer Science*, 21, 2025. doi:10.46298/lmcs-21(2:5)2025.
7. Paula Blechschmidt, Kirstin Peters, and Uwe Nestmann. Compositional interface refinement through subtyping in probabilistic session types. In Zhiming Liu, Adnane Saoud, and Heike Wehrheim, editors, *Theoretical Aspects of Computing - ICTAC 2025 - 22nd International Colloquium, Marrakech, Morocco, November 24-28, 2025, Proceedings*, Lecture Notes in Computer Science, pages 145–163. Springer, 2025. doi:10.1007/978-3-032-11176-0_10.

8. Laura Bocchi, Raymond Hu, Adriana Laura Voinea, and Simon J. Thompson. Mixed choice in asynchronous multiparty session types. *CoRR*, abs/2602.23927, 2026. To appear in OOPSLA'26. URL: `https://doi.org/10.48550/arXiv.2602.23927`, `arXiv:2602.23927`, `doi:10.48550/ARXIV.2602.23927`.
9. Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, April 1983.
10. Mario Bravetti, Luca Padovani, and Gianluigi Zavattaro. A sound and complete characterization of fair asynchronous session subtyping. In Patricia Bouyer and Jaco van de Pol, editors, *36th International Conference on Concurrency Theory, CONCUR 2025, Aarhus, Denmark, August 26-29, 2025*, LIPIcs, pages 11:1–11:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. URL: `https://doi.org/10.4230/LIPIcs.CONCUR.2025.11`, `doi:10.4230/LIPICS.CONCUR.2025.11`.
11. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8:1–8:78, 2012. `doi:10.1145/2220365.2220367`.
12. Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. *Logical Methods in Computer Science*, Volume 8, Issue 1, Mar 2012. URL: `https://lmcs.episciences.org/773`, `doi:10.2168/LMCS-8(1:24)2012`.
13. Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the Preciseness of Subtyping in Session Types. *Logical Methods in Computer Science*, 13:1–62, 2017. `doi:10.23638/LMCS-13(2:12)2017`.
14. Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. On the Preciseness of Subtyping in Session Types: 10 Years Later. In *Principles and Practice of Declarative Programming*, pages –. ACM, 2024. `doi:10.1145/3678232.3678258`.
15. Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. A Gentle Introduction to Multiparty Asynchronous Session Types. In *15th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Multicore Programming*, volume 9104 of *LNCS*, pages 146–178. Springer, 2015.
16. Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *Mathematical Structures in Computer Science*, 26:238–302, 2015. `doi:10.1017/S0960129514000188`.
17. Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Communications in choreographies, revisited. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, page 1248–1255, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3167132.3167267`.
18. Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*, volume 6901 of *Lecture Notes in Computer Science*, pages 280–296. Springer, 2011. `doi:10.1007/978-3-642-23217-6\_19`.
19. Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty Session Types Meet Communicating Automata. In *21st European Symposium on Programming*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012. `doi:10.1007/978-3-642-28869-2_10`.
20. Mariangiola Dezani-Ciancaglini and Silvia Ghilezan. Preciseness of subtyping on intersection and union types. In *Lecture Notes in Computer Science*, Lecture Notes

in Computer Science, pages 194–207. Springer International Publishing, Cham, 2014.
21. Cinzia Di Giusto, Etienne Lozes, and Pascal Urso. Realisability and complementability of multiparty session types. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*, PPDP '25, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3756907.3756918.
22. Cinzia Di Giusto, Etienne Lozes, and Pascal Urso. Realisability and complementability of multiparty session types. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*, PPDP '25, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3756907.3756918.
23. Silvia Ghilezan, Svetlana Jakšić, Jovanka Pantović, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *Journal of Logical and Algebraic Methods in Programming*, 2018. doi:10.1016/j.jlamp.2018.12.002.
24. Silvia Ghilezan, Jovanka Pantović, Ivan Prokić, Alceste Scalas, and Nobuko Yoshida. Precise Subtyping for Asynchronous Multiparty Sessions. *ACM Transactions on Computational Logic*, Volume 24, Issue 2(14):1–73, 2023. doi:10.1145/3568422.
25. Ruben Hamers, Erik Horlings, and Sung-Shik Jongmans. The discourje project: run-time verification of communication protocols in clojure. *Int. J. Softw. Tools Technol. Transf.*, 24(5):757–782, October 2022.
26. Jonas Kastberg Hinrichsen, Iwan Quémerais, and Lars Birkedal. Mixtris: Mechanised higher-order separation logic for mixed choice multiparty message passing. *Proc. ACM Program. Lang.*, 10(OOPSLA1), April 2026. doi:10.1145/3798224.
27. Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, pages 509–523, Berlin, Heidelberg, 1993. Springer. doi:10.1007/3-540-57208-2_35.
28. Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In Chris Hankin, editor, *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*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFB0053567.
29. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 273–284. ACM, 2008. doi:10.1145/1328897.1328472.
30. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *Journal of the ACM*, 63:1–67, 2016. doi:10.1145/2827695.
31. Raymond Hu and Nobuko Yoshida. Hybrid Session Verification through Endpoint API Generation. In *19th International Conference on Fundamental Approaches to Software Engineering*, volume 9633 of *LNCS*, pages 401–418, Berlin, Heidelberg, 2016. Springer. doi:10.1007/978-3-662-49665-7_24.
32. Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In *22nd European Conference on Object-Oriented Programming*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008. doi:10.1007/978-3-540-70592-5_22.

33. Hajime Ishihara and Toshihiko Kurata. Completeness of intersection and union type assignment systems for call-by-value λ-models. *Theor. Comput. Sci.*, 272(1-2):197–221, February 2002.
34. Sung-Shik Jongmans and Nobuko Yoshida. Exploring Type-Level Bisimilarity towards More Expressive Multiparty Session Types. In *29th European Symposium on Programming*, volume 12075 of *LNCS*, pages 251–279. Springer, 2020. doi:10.1007/978-3-030-44914-810.
35. Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Stay safe under panic: Affine rust programming with multiparty session types. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, volume 222 of *LIPIcs*, pages 4:1–4:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2022.4, doi:10.4230/LIPICS.ECOOP.2022.4.
36. Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 221–232. ACM, 2015. doi:10.1145/2676726.2676964.
37. Elaine Li, Felix Stutz, and Thomas Wies. Deciding subtyping for asynchronous multiparty sessions. In *Lecture Notes in Computer Science*, Lecture notes in computer science, pages 176–205. Springer Nature Switzerland, Cham, 2024.
38. Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. Complete multiparty session type projection with automata. In *Lecture Notes in Computer Science*, Lecture Notes in Computer Science, pages 350–373. Springer Nature Switzerland, Cham, 2023.
39. Jay Ligatti, Jeremy Blackburn, and Michael Nachtigal. On subtyping-relation completeness, with an application to iso-recursive types. *ACM Trans. Program. Lang. Syst.*, 39(1), March 2017. doi:10.1145/2994596.
40. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994. doi:10.1145/197320.197383.
41. Rupak Majumdar, Madhavan Mukund, Felix Stutz, and Damien Zufferey. Generalising Projection in Asynchronous Multiparty Session Types. In Serge Haddad and Daniele Varacca, editors, *32nd International Conference on Concurrency Theory (CONCUR 2021)*, volume 203 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:24, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CONCUR.2021.35, doi:10.4230/LIPIcs.CONCUR.2021.35.
42. Jake Masters and Nobuko Yoshida. Mixed choice multiparty session types, precisely, 2026. URL: https://arxiv.org/abs/2608.10704, arXiv:2608.10704.
43. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, September 1992.
44. Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. SPY: Local Verification of Global Protocols. In *4th International Conference on Runtime Verification*, volume 8174 of *LNCS*, pages 363–358. Springer, 2013. doi:10.1007/978-3-642-40787-1_25.
45. Luca Padovani, Vasco Thudichum Vasconcelos, and Hugo Torres Vieira. Typing liveness in multiparty communicating systems. In *Lecture Notes in Computer Science*, Lecture Notes in Computer Science, pages 147–162. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

46. Luca Padovani and Gianluigi Zavattaro. Fair termination of asynchronous binary sessions. In Jonathan Aldrich and Alexandra Silva, editors, *39th European Conference on Object-Oriented Programming, ECOOP 2025, Bergen, Norway, June 30 - July 2, 2025*, LIPIcs, pages 24:1–24:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. URL: `https://doi.org/10.4230/LIPIcs.ECOOP.2025.24`, `doi:10.4230/LIPICS.ECOOP.2025.24`.
47. Catuscia Palamidessi. Comparing the expressive power of the synchronous and asynchronous pi-calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003.
48. Kirstin Peters and Nobuko Yoshida. Separation and encodability in mixed choice multiparty sessions. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '24, New York, NY, USA, 2024. Association for Computing Machinery. `doi:10.1145/3661814.3662085`.
49. Kai Pischke, Jake Masters, and Nobuko Yoshida. Asynchronous global protocols, precisely. *Inf. Comput.*, (105501):105501, August 2026.
50. Kai Pischke and Nobuko Yoshida. Top Down=Bottom Up: Sound and Complete Characterisations of Liveness by Multiparty Global Protocols. *Proc. ACM Program. Lang.*, 2026.
51. Ivan Prokić, Simona Prokić, Silvia Ghilezan, Alceste Scalas, and Nobuko Yoshida. On Asynchronous Multiparty Session Types for Federated Learning. In *22st International Colloquium on Theoretical Aspects of Computing, 2025, Marrakech, Morocco*, volume 16237 of *Lecture Notes in Computer Science*, pages 164–182. Springer Nature, 2025. `doi:10.1007/978-3-032-11176-0_11`.
52. Alceste Scalas and Nobuko Yoshida. Less is More: Multiparty Session Types Revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, January 2019. `doi:10.1145/3290343`.
53. Felix Stutz and Emanuele D'Osualdo. An automata-theoretic basis for specification and type checking of multiparty protocols. In *Programming Languages and Systems: 34th European Symposium on Programming, ESOP 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3–8, 2025, Proceedings, Part II*, page 314–346, Berlin, Heidelberg, 2025. Springer-Verlag. `doi:10.1007/978-3-031-91121-7_13`.
54. Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 1994. `doi:10.1007/3-540-58184-7_118`.
55. Thien Udomsrirungruang and Nobuko Yoshida. Top-Down or Bottom-Up? Complexity Analyses of Synchronous Multiparty Session Types. *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages –, 2025.
56. Steffen van Bakel, Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Yoko Motohama. The minimal relevant logic and the call-by-value lambda calculus. Technical report, Technical Report TR-ARP-05-2000, The Australian National University, 2000.
57. Bas van den Heuvel and Jorge A Pérez. A decentralized analysis of multiparty protocols. *Sci. Comput. Program.*, 222(102840):102840, October 2022.
58. Jérôme Vouillon. Subtyping union types. In *Computer Science Logic*, Lecture Notes in Computer Science, pages 415–429. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

59. Nobuko Yoshida. Programming Language Implementations with Multiparty Session Types. In *Active Object Languages: Current Research Trends*, pages 147–165. Springer Nature Switzerland, 2024. doi:10.1007/978-3-031-51060-1_6.
60. Nobuko Yoshida and Lorenzo Gheri. A very gentle introduction to multiparty session types. In Dang Van Hung and Meenakshi D'Souza, editors, *Distributed Computing and Internet Technology - 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings*, volume 11969 of *Lecture Notes in Computer Science*, pages 73–93. Springer, 2020. doi:10.1007/978-3-030-36987-3_5.
61. Nobuko Yoshida and Ping Hou. Less is More Revisited: Association with Global Multiparty Session Types. In *The Practice of Formal Methods: Essays in Honour of Cliff Jones, Part II*, pages –. LNCS, 2024.