Fair Termination of Multiparty Sessions

Luca Ciccone
University of Torino, Italy

Francesco Dagnino
University of Genova, Italy

Luca Padovani
University of Torino, Italy

Abstract

There exists a broad family of multiparty sessions in which the progress of one session participant is not unconditional, but depends on the choices performed by other participants. These sessions fall outside the scope of currently available session type systems that guarantee progress. In this work we propose the first type system ensuring that well-typed multiparty sessions, including those exhibiting the aforementioned dependencies, fairly terminate. Fair termination is termination under a fairness assumption that disregards those interactions deemed unfair and therefore unrealistic. Fair termination, combined with the usual safety properties ensured within sessions, not only is desirable *per se*, but it entails progress and enables a compositional form of static analysis such that the well-typed composition of fairly terminating sessions results in a fairly terminating program.

2012 ACM Subject Classification Theory of computation → Process calculi; Theory of computation → Type structures; Theory of computation → Program analysis

Keywords and phrases Multiparty sessions, fair termination, fair subtyping, deadlock freedom

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.26

Related Version *Full Version*: `https://arxiv.org/abs/2205.08786` [10]

Funding The second author was partially supported by the MUR project T-LADIES (PRIN 2020TL3X8X).

Acknowledgements We are grateful to the anonymous ECOOP reviewers for their thoughtful and useful comments that helped us improving form and content of the paper.

1 Introduction

Sessions [24, 25, 27] are private conversations among processes following a protocol specification called session type. The decomposition of a distributed program into sessions enables its modular static analysis and the enforcement of useful properties through a type system. Examples of such properties are *communication safety* (no message of the wrong type is ever exchanged), *protocol fidelity* (messages are exchanged in the order prescribed by session types) and *deadlock freedom* (the program keeps running unless all sessions have terminated). These are all instances of *safety properties*, implying that "nothing bad" happens. In general, one is also interested in reasoning and possibly enforcing *liveness properties*, those implying that "something good" happens [39]. Examples of liveness properties are *junk freedom* (every message is eventually received), *progress* (every non-terminated participant of a session eventually performs an action) and *termination* (every session eventually comes to an end).

An enduring limitation of current type systems for multiparty sessions is that *they ensure progress for any participant of a session only when such progress can be established independently of the choices performed by the other participants.* To illustrate the impact of this limitation, consider a session made of three participants named buyer, seller and carrier

© Luca Ciccone, Francesco Dagnino, and Luca Padovani;
licensed under Creative Commons License CC-BY 4.0

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 26; pp. 26:1–26:26
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

in which the buyer aims at purchasing an unspecified number of items from the seller and the seller relies on a carrier for delivering the purchased items to the buyer. The buyer behaves according to the session type S that satisfies the equation

$$S = \mathsf{seller}!\mathsf{add}.S + \mathsf{seller}!\mathsf{pay}.!\mathsf{end} \tag{1}$$

indicating that it either pays the seller or it adds an item to the shopping cart and then repeats the same behavior. In this session type, add and pay are messages targeted to the participant with role seller. In turn, the seller accepts add messages from the buyer until a pay message is received, at which point it instructs the carrier to ship the items. Thus, its behavior is described by the session type T that satisfies the equation

$$T = \mathsf{buyer}?\mathsf{add}.T + \mathsf{buyer}?\mathsf{pay}.\mathsf{carrier}!\mathsf{ship}.!\mathsf{end} \tag{2}$$

Finally, the carrier just waits for the ship message from the seller. So, its behavior is described by the session type

$$\mathsf{seller}?\mathsf{ship}.?\mathsf{end} \tag{3}$$

No available type system is able to guarantee progress for every participant of this multiparty session. What makes this session somewhat difficult to reason about is that *the progress of the carrier is not unconditional but depends on the choices performed by the buyer*: the carrier can make progress only if the buyer eventually pays the seller.

In this work we propose a type system that guarantees the *fair termination* of sessions, that is termination under a *fairness assumption*. The assumption we make is an instance of *relative fairness* [45] and can be roughly spelled out as follows:

If termination is always possible, then it is inevitable. (4)

The multiparty session sketched above terminates under this fairness assumption: since it is always possible for the buyer to pay the seller and terminate, in every fair execution of the session the buyer eventually pays the seller, even though we do not know (nor do we impose) an upper bound to the number of items that the buyer may add to the shopping cart. Simply, the non-terminating execution of the session in which the buyer keeps adding items to the shopping cart but never pays is assumed unrealistic and so it can be ignored insofar as termination is concerned.

The reader might wonder why we focus on fair termination instead of considering some fair version of progress. There are three reasons why we think that fair termination is overall more appropriate than just progress. First of all, ensuring that sessions (fairly) terminate is consistent with the usual interpretation of the word "session" as an activity that lasts for a *finite amount of time*, even when the maximum duration of the activity is not known *a priori*. Second, *fair termination implies progress* when it is guaranteed along with the usual safety properties of sessions. Indeed, if the session eventually terminates, it must be the case that any non-terminated participant (think of the carrier waiting for a ship message) is guaranteed to eventually make progress, even when such progress *depends* on choices made by other participants (like the buyer sending pay to the seller). Last but not least, *fair session termination enables compositional reasoning* in the presence of multiple sessions. This is not true for progress: if an action on a session s is blocked by actions on a different session t, then knowing that the session t enjoys progress does not necessarily guarantee that the action on s will eventually be performed (the interaction on t might continue forever). On the contrary, knowing that t fairly terminates guarantees that the action on s will eventually be scheduled and performed, so that s may in turn progress towards termination.

Remarkably, the fairness assumption alone does not suffice to turn any multiparty session type system into one that ensures fair termination. In fact, there are several sources of potentially non-terminating behaviors that must be ruled out in well-typed processes:

1. Fairly terminating (and even finite) sessions may be chained, nested, interleaved in such a way that some pending activities are postponed forever. To avoid this problem, our type system makes sure that the effort required by a well-typed process in order to terminate remains finite. At the same time, it does not (always) prevent the modeling of processes that create an unbounded number of sessions.
2. The type-level constraints usually imposed to well-typed sessions – *duality* [24, 25, 27], *liveness* [46], *coherence* [9], just to mention a few – are in general too weak to entail fair session termination. Our type system adopts a stronger notion of "correct multiparty session" that entails fair termination. Variants of this notion have already appeared in the literature [5, 42], but we use it here for the first time to relate types and processes.
3. A certain mismatch is usually allowed between the structure of session types and the structure of the processes that adhere to those types. This mismatch is formalized by a subtyping relation for session types which, in its standard formulation [23], may introduce non-terminating behaviors. Our type system adopts *fair subtyping* [42], a liveness-preserving refinement of the standard subtyping relation for session types [23].

Summary of contributions. We present the first type system ensuring the fair termination of multiparty sessions and capable of addressing a number of natural communication patterns that are out of scope of existing multiparty session type systems [46, 48]. We exploit the compositional reasoning enabled by fair termination to prove a strong soundness result whereby a well-typed composition of fairly terminating sessions is a fairly terminating program (Theorem 5.4). This result scales smoothly also in presence of session chaining, session nesting, session interleaving, session delegation and dynamic session creation. In sharp contrast, the liveness properties ensured by previous multiparty session type systems are either limited to single-session programs [46, 48] or require a richer type structure [43, 15]. Our contributions extend and generalize previous work on the fair termination of binary sessions [14] and allow for the modeling of (intra-session) cyclic network topologies and of multiparty sessions that cannot be decomposed into equivalent (well-typed) binary sessions. Decidability of type checking is not substantially more difficult than the same problem in the binary setting [14]. *En passant*, in this paper we also provide a new characterization of fair subtyping for (multiparty) session types (Table 5) that is substantially simpler than those appearing in previous works [40, 42, 13, 14].

Structure of the paper. We recall the key notions related to fair termination (Section 2) before presenting our language of multiparty sessions (Section 3). Then, we define multiparty session types and fair subtyping (Section 4) and present the typing rules and the soundness properties of the type system (Section 5). In the latter part of the paper we illustrate a few more advanced examples of well-typed processes (Section 6), we discuss related work in more detail (Section 7) and we provide hints at further developments (Section 8).

2 Fair Termination

Since the notion of fair termination will apply to several different entities (session types, multiparty sessions, processes) here we define it for a generic reduction system. Later on we will show various instantiations of this definition. A *reduction system* is a pair $(\mathcal{S}, \rightarrow)$ where

$\mathcal{S}$ is a set of *states* and $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is a *reduction relation*. We adopt the following notation: we let C and D range over states; we write $C \rightarrow$ if there exists $D \in \mathcal{S}$ such that $C \rightarrow D$; we write $C \not\rightarrow$ if not $C \rightarrow$; we write $\Rightarrow$ for the reflexive, transitive closure of $\rightarrow$. We say that D is *reachable* from C if $C \Rightarrow D$.

As an example, the reduction system $(\{A, B\}, \{(A, A), (A, B)\})$ models an entity that can be in two states, A or B, and such that the entity may perform a reduction to remain in state A or a reduction to move from state A to state B. To formalize the evolution of an entity from a particular state we define *runs*.

▶ **Definition 2.1** (runs and maximal runs). *A* run *of C is a (finite or infinite) sequence $C_0 C_1 \dots C_i \dots$ of states such that $C_0 = C$ and $C_i \rightarrow C_{i+1}$ for every valid i. A run is* maximal *if either it is infinite or if its last state C_n is such that $C_n \not\rightarrow$.*

Hereafter we let ρ range over runs. Each run in the previously defined reduction system is either of the form A^n – a finite sequence of A – or of the form $A^n B$ – a finite sequence of A followed by one B – or A^ω – an infinite sequence of A. Among these, the runs of the form $A^n B$ and A^ω are maximal, whereas no run of the form A^n is maximal.

We now use runs to define different termination properties of states: we say that C is *weakly terminating* if there exists a maximal run of C that is finite; we say that C is *terminating* if every maximal run of C is finite; we say that C is *diverging* if every maximal run of C is infinite. *Fair termination* [21] is a termination property that only considers a subset of all (maximal) runs of a state, those that are considered to be “realistic” or “fair” according to some fairness assumption. The assumption that we make in this work, and that we stated in words in (4), is formalized thus:

▶ **Definition 2.2** (fair run). *A run is* fair *if it contains finitely many weakly terminating states. Conversely, a run is* unfair *if it contains infinitely many weakly terminating states.*

Continuing with the previous example, the runs of the form A^n and $A^n B$ are fair, whereas the run A^ω is unfair. In general, an unfair run is an execution in which termination is always within reach, but is never reached.

A key requirement of any fairness assumption is that it must be possible to extend every finite run to a maximal fair one. This property is called *feasibility* [4, 47] or *machine closure* [37]. It is easy to see that our fairness assumption is feasible:

▶ **Lemma 2.3.** *If ρ is a finite run, then there exists ρ' such that $\rho\rho'$ is a maximal fair run.*

Fair termination is finiteness of all maximal fair runs:

▶ **Definition 2.4** (fair termination). *We say that C is* fairly terminating *if every maximal fair run of C is finite.*

In the reduction system given above, A is fairly terminating. Indeed, all the maximal runs of the form $A^n B$ are finite whereas A^ω, which is the only infinite fair run of A, is unfair.

For the particular fairness assumption that we make, it is possible to provide a sound and complete characterization of fair termination that does not mention fair runs. This characterization will be useful to relate fair termination with the notion of correct multiparty session (Definition 4.2) and the soundness property of the type system (Theorem 5.4).

▶ **Theorem 2.5.** *Let $(\mathcal{S}, \rightarrow)$ be a reduction system and $C \in \mathcal{S}$. Then C is fairly terminating if and only if every state reachable from C is weakly terminating.*

Table 1 Syntax of processes.

P, Q, R	$::=$		**Process**			
		done	termination	$\mid$	$A\langle\overline{u}\rangle$	invocation
	$\mid$	$\mathsf{wait}\,u.P$	signal input	$\mid$	$\mathsf{close}\,u$	signal output
	$\mid$	$u[\mathsf{p}]?(x).P$	channel input	$\mid$	$u[\mathsf{p}]!v.P$	channel output
	$\mid$	$u[\mathsf{p}]\pi\{\mathsf{m}_i.P_i\}_{i\in I}$	tag input/output	$\mid$	$P \oplus Q$	choice
	$\mid$	$(s)(P_1 \mid \cdots \mid P_n)$	session	$\mid$	$\lceil u \rceil P$	cast

▶ Remark 2.6 (fair reachability of predicates [45]). Most fairness assumptions have the form "if *something* is infinitely often possible then *something* happens infinitely often" and, in this respect, our formulation of fair run (Definition 2.2) looks slightly unconventional. However, it is not difficult to realize that Definition 2.2 is an instance of the notion of fair reachability of predicates as defined by Queille and Sifakis [45, Definition 3]. According to Queille and Sifakis, a run ρ is fair with respect to some predicate $\mathcal{C} \subseteq \mathcal{S}$ if, whenever in ρ there are infinitely many states from which a state in $\mathcal{C}$ is reachable, then in ρ there are infinitely many occurrences of states in $\mathcal{C}$. When we take $\mathcal{C}$ to be $\nrightarrow$, that is the set of terminated states that do not reduce, pretending that irreducible states should occur infinitely often in the run is nonsensical. So, the fairness assumption boils down to assuming that such states should *not* be reachable infinitely often, which is precisely the formulation of Definition 2.2. ⌟

3 A Calculus of Multiparty Sessions

In this section we define the calculus for multiparty sessions on which we apply our static analysis technique. The calculus is an extension of the one presented by Ciccone and Padovani [14] to multiparty sessions in the style of Scalas and Yoshida [46].

We use an infinite set of *variables* ranged over by x, y, z, an infinite set of *session names* ranged over by s and t, a set of *roles* ranged over by p, q, r, a set of *message tags* ranged over by m, and a set of *process names* ranged over by A, B, C. In the literature of sessions tags are usually called labels. We adopt a different terminology to avoid confusion with another notion of label that we introduce in Section 4. We use roles to distinguish the participants of a session. In particular, an *endpoint* $s[\mathsf{p}]$ consists of a session name s and a role p and is used by the participant with role p to interact with the other participants of the session s. We use u and v to range over *channels*, which are either variables or session endpoints. We write $\overline{x}$ and $\overline{u}$ to denote possibly empty sequences of variables and channels, extending this notation to other entities. We use π to range over the elements of the set $\{?, !\}$ of *polarities*, distinguishing input actions ($?$) from output actions ($!$).

A *program* is a finite set of *definitions* of the form $A(\overline{x}) \triangleq P$, at most one for each process name, where P is a term generated by the syntax shown in Table 1. The term done denotes the terminated process that performs no action. The term $A\langle\overline{u}\rangle$ denotes the invocation of the process with name A passing the channels $\overline{u}$ as arguments. When $\overline{u}$ is empty we just write A instead of $A\langle\rangle$. The term $\mathsf{close}\,u$ denotes the process that sends a termination signal on the channel u, whereas $\mathsf{wait}\,u.P$ denotes the process that waits for a termination signal from channel u and then continues as P. The term $u[\mathsf{p}]!v.P$ denotes the process that sends the channel v on the channel u to the role p and then continues as P. Dually, $u[\mathsf{p}]?(x).P$ denotes the process that receives a channel from the role p on the channel u and then continues as P where x is replaced with the received channel. The term $u[\mathsf{p}]\pi\{\mathsf{m}_i.P_i\}_{i\in I}$ denotes a process that exchanges one of the tags m_i on the channel u with the role p and then continues as P_i.

Table 2 Structural precongruence of processes.

[s-par-comm]	$(s)(\overline{P} \mid P \mid Q \mid \overline{Q}) \preccurlyeq (s)(\overline{P} \mid Q \mid P \mid \overline{Q})$	
[s-par-assoc]	$(s)(\overline{P} \mid (t)(R \mid \overline{Q})) \preccurlyeq (t)((s)(\overline{P} \mid R) \mid \overline{Q})$	if $s \in \mathsf{fn}(R)$
[s-cast-comm]	$\lceil u \rceil \lceil v \rceil P \preccurlyeq \lceil v \rceil \lceil u \rceil P$	
[s-cast-new]	$(s)(\lceil s[\mathsf{p}] \rceil P \mid \overline{Q}) \preccurlyeq (s)(P \mid \overline{Q})$	
[s-cast-swap]	$(s)(\lceil t[\mathsf{p}] \rceil P \mid \overline{Q}) \preccurlyeq \lceil t[\mathsf{p}] \rceil (s)(P \mid \overline{Q})$	if $s \neq t$
[s-call]	$A\langle \overline{u} \rangle \preccurlyeq P\{\overline{u}/\overline{x}\}$	if $A(\overline{x}) \triangleq P$

Whether the tag is sent or received depends on the polarity π and, as it will be clear from the operational semantics, the polarity π also determines whether the process behaves as an internal choice (when π is !) or an external choice (when π is ?). In the first case the process chooses *actively* the tag being sent, whereas in the second case the process reacts *passively* to the tag being received. We assume that I is finite and non-empty and also that the tags m_i are pairwise distinct. For brevity, we write $u[\mathsf{p}]\pi\mathsf{m}_k.P_k$ instead of $u[\mathsf{p}]\pi\{\mathsf{m}_i.P_i\}_{i\in I}$ when I is the singleton set $\{k\}$. The term $P \oplus Q$ denotes a process that non-deterministically behaves either as P or as Q.

A term $(s)(P_1 \mid \cdots \mid P_n)$ with $n \geq 1$ denotes the parallel composition of n processes, each of them being a participant of the session s. Each process is associated with a distinct a role p_i and communicates in s through the endpoint $s[\mathsf{p}_i]$. Combining session creation and parallel composition in a single form is common in session type systems based on linear logic [6, 49, 38] and helps guaranteeing deadlock freedom. Finally, a *cast* $\lceil u \rceil P$ denotes a process that behaves exactly as P. This form is only relevant for the type system (Section 5) and denotes the fact that the type of u is subject to an application of subtyping.

The free and bound names of a process are defined as usual, the latter ones being easily recognizable as they occur within round parenteses. We write $\mathsf{fn}(P)$ for the set of free names of P and we identify processes modulo renaming of bound names. Note that $\mathsf{fn}(P)$ may contain variables and session names, but not endpoints. Occasionally we write $A(\overline{x}) \triangleq P$ as a predicate or side condition, meaning that P is the process associated with the process name A. For each of such definitions we assume that $\mathsf{fn}(P) \subseteq \{\overline{x}\}$.

The operational semantics of processes is given by the structural precongruence relation $\preccurlyeq$ defined in Table 2 and the reduction relation $\rightarrow$ defined in Table 3. As usual, structural precongruence allows us to rearrange the structure of processes without altering their meaning, whereas reduction expresses an actual computation or interaction step. The adoption of a structural *pre*congruence (as opposed to a more common congruence relation) is not strictly necessary, but it simplifies the technical development by reducing the number of cases we have to consider in proofs without affecting the properties of the calculus in any way.

Rules [s-par-comm] and [s-par-assoc] state commutativity and associativity of parallel composition of processes (we write $\overline{P}$ to denote possibly empty parallel compositions of processes). In [s-par-assoc], the side condition $s \in \mathsf{fn}(R)$ makes sure that R is indeed a participant of the session s. Note that this rule only states right-to-left associativity. Left-to-right associativity is derivable from this rule and repeated uses of [s-par-comm]. Rule [s-cast-comm] allows us to swap two consecutive casts. Rule [s-cast-new] removes an unguarded cast on an endpoint of the restricted session (we refer to this operation as “performing the cast”). Rule [s-cast-swap] swaps a cast and a restricted session as long as the endpoint in the cast refers to a different session. Finally, rule [s-call] unfolds a process invocation to its definition. Hereafter, we write $\{u/x\}$ for the capture-avoiding substitution of each free occurrence of x with u and $\{\overline{u}/\overline{x}\}$ for its natural extension to equal-length tuples

Table 3 Reduction of processes.

$$\text{[R-CHOICE]}\ \frac{}{P_1 \oplus P_2 \to P_k}\ k \in \{1,2\} \qquad \text{[R-SIGNAL]}\ \frac{}{(s)(\mathsf{wait}\, s[\mathsf{p}].P \mid \mathsf{close}\, s[\mathsf{q}_1] \mid \cdots \mid \mathsf{close}\, s[\mathsf{q}_n]) \to P}$$

$$\text{[R-CHANNEL]}\ \frac{}{(s)(s[\mathsf{p}][\mathsf{q}]!v.P \mid s[\mathsf{q}][\mathsf{p}]?(x).Q \mid \overline{R}) \to (s)(P \mid Q\{v/x\} \mid \overline{R})}$$

$$\text{[R-PICK]}\ \frac{}{(s)(s[\mathsf{p}][\mathsf{q}]!\{\mathsf{m}_i.P_i\}_{i\in I} \mid \overline{Q}) \to (s)(s[\mathsf{p}][\mathsf{q}]!\mathsf{m}_k.P_k \mid \overline{Q})}\ k \in I$$

$$\text{[R-TAG]}\ \frac{}{(s)(s[\mathsf{p}][\mathsf{q}]!\mathsf{m}_k.P \mid s[\mathsf{q}][\mathsf{p}]?\{\mathsf{m}_i.Q_i\}_{i\in I} \mid \overline{R}) \to (s)(P \mid Q_k \mid \overline{R})}\ k \in I$$

$$\text{[R-PAR]}\ \frac{P \to Q}{(s)(P \mid \overline{R}) \to (s)(Q \mid \overline{R})} \qquad \text{[R-CAST]}\ \frac{P \to Q}{\lceil u \rceil P \to \lceil u \rceil Q} \qquad \text{[R-STRUCT]}\ \frac{P \preccurlyeq P' \qquad P' \to Q' \qquad Q' \preccurlyeq Q}{P \to Q}$$

of variables and names. The rules [S-CAST-NEW], [S-CAST-SWAP] and [S-CALL] are not invertible: by [S-CAST-NEW] casts can only be removed but never added; by [S-CAST-SWAP] casts can only be moved closer to their restriction, so that they can be eventually performed by [S-CAST-NEW]; by [S-CALL] process invocations can only be unfolded.

The reduction relation is quite standard. Rule [R-CHOICE] reduces $P_1 \oplus P_2$ to either P_1 or P_2, non deterministically. Rule [R-SIGNAL] terminates a session in which all participants ($\mathsf{q}_1, \dots, \mathsf{q}_n$) but one ($\mathsf{p}$) are sending a termination signal and p is waiting for it; the resulting process is the continuation of the participant p. Rule [R-CHANNEL] models the exchange of a channel among two participants of a session. Rule [R-PICK] models an internal choice whereby a process picks one particular tag m_k to send on a session. Rule [R-TAG] synchronizes two participants p and q on the tag chosen by p. Finally, rules [R-PAR], [R-CAST] and [R-STRUCT] close reductions under parallel compositions and casts and by structural precongruence.

In the rest of this section we illustrate the main features of the calculus with some examples. For none of them the existing multiparty session type systems are able to guarantee progress.

▶ **Example 3.1** (purchase). We model a particular instance of the buyer-seller-carrier interaction that we have informally discussed in Section 1 with the following definitions:

$$\begin{aligned}
\mathit{Main} &\triangleq (s)(\mathit{Buyer}\langle s[\mathsf{buyer}]\rangle \mid \mathit{Seller}\langle s[\mathsf{seller}]\rangle \mid \mathit{Carrier}\langle s[\mathsf{carrier}]\rangle)\\
\mathit{Buyer}(x) &\triangleq x[\mathsf{seller}]!\{\mathsf{add}.x[\mathsf{seller}]!\mathsf{add}.\mathit{Buyer}\langle x\rangle, \mathsf{pay}.\mathsf{close}\, x\}\\
\mathit{Seller}(x) &\triangleq x[\mathsf{buyer}]?\{\mathsf{add}.\mathit{Seller}\langle x\rangle, \mathsf{pay}.x[\mathsf{carrier}]!\mathsf{ship}.\mathsf{close}\, x\}\\
\mathit{Carrier}(x) &\triangleq x[\mathsf{seller}]?\mathsf{ship}.\mathsf{wait}\, x.\mathsf{done}
\end{aligned}$$

Note that the buyer either sends pay or it sends two add messages in a row before repeating this behavior. That is, this particular buyer always adds an even number of items to the shopping cart. Nonetheless, the buyer periodically has a chance to send a pay message and terminate. Therefore, the execution of the program in which the buyer only sends add is unfair according to Definition 2.2 hence this program is fairly terminating. ⌟

▶ **Example 3.2** (purchase with negotiation)**.** Consider a variation of Example 3.1 in which the buyer, before making the payment, negotiates with a secondary buyer for an arbitrarily long time. The interaction happens in two nested sessions, an outer one involving the primary buyer, the seller and the carrier, and an inner one involving only the two buyers. We model the interaction as the program below, in which we collapse role names to their initials.

$$\begin{aligned}
\mathit{Main} &\triangleq (s)(\mathit{Buyer}\langle s[\mathsf{b}]\rangle \mid \mathit{Seller}\langle s[\mathsf{s}]\rangle \mid \mathit{Carrier}\langle s[\mathsf{c}]\rangle)\\
\mathit{Buyer}(x) &\triangleq x[\mathsf{s}]!\mathsf{query}.x[\mathsf{s}]?\mathsf{price}.(t)(\mathit{Buyer}_1\langle x, t[\mathsf{b}_1]\rangle \mid \mathit{Buyer}_2\langle t[\mathsf{b}_2]\rangle)\\
\mathit{Seller}(x) &\triangleq x[\mathsf{b}]?\mathsf{query}.x[\mathsf{b}]!\mathsf{price}.x[\mathsf{b}]?\{\mathsf{pay}.x[\mathsf{c}]!\mathsf{ship}.\mathsf{close}\,x, \mathsf{cancel}.x[\mathsf{c}]!\mathsf{cancel}.\mathsf{close}\,x\}\\
\mathit{Carrier}(x) &\triangleq x[\mathsf{s}]?\{\mathsf{ship}.x[\mathsf{b}]!\mathsf{box}.\mathsf{close}\,x, \mathsf{cancel}.\mathsf{close}\,x\}\\
\mathit{Buyer}_1(x,y) &\triangleq y[\mathsf{b}_2]!\{\mathsf{split}.y[\mathsf{b}_2]?\{\mathsf{yes}.\lceil x\rceil x[\mathsf{s}]!\mathsf{ok}.x[\mathsf{c}]?\mathsf{box}.\mathsf{wait}\,x.\mathsf{wait}\,y.\mathsf{done},\\
&\qquad\qquad\qquad\qquad\qquad \mathsf{no}.\mathit{Buyer}_1\langle x,y\rangle\},\\
&\qquad\qquad\quad \mathsf{giveup}.\mathsf{wait}\,y.\lceil x\rceil x[\mathsf{s}]!\mathsf{cancel}.\mathsf{wait}\,x.\mathsf{done}\}\\
\mathit{Buyer}_2(y) &\triangleq y[\mathsf{b}_1]?\{\mathsf{split}.y[\mathsf{b}_1]!\{\mathsf{yes}.\mathsf{close}\,y, \mathsf{no}.\mathit{Buyer}_2\langle y\rangle\}, \mathsf{giveup}.\mathsf{close}\,y\}
\end{aligned}$$

The buyer queries the seller which replies with a price. At this point, *Buyer* creates a new session t and forks as a primary buyer Buyer_1 and a secondary buyer Buyer_2. The interaction between the two sub-buyers goes on until either Buyer_1 gives up or Buyer_2 accepts its share of the price. In the former case, the primary buyer waits for the internal session to terminate and cancels the order with the seller which, in turn, aborts the transaction with the carrier. In the latter case, the buyer confirms the order to the seller, which then instructs the carrier to ship a box to the buyer.

Note that the outermost session s, taken in isolation, terminates in a bounded number of interactions, but its progress cannot be established without assuming that the innermost session t terminates. In particular, if the two buyers keep negotiating forever, the seller and the carrier starve. However, the innermost session can terminate if Buyer_1 sends giveup to Buyer_2 or if Buyer_2 sends yes to Buyer_1. Thus, the run in which the two buyers negotiate forever is unfair, the session t fairly terminates and the session s terminates as well.

On the technical side, note that the definition of Buyer_1 contains two casts on the variable x. As we will see in Example 6.1, these casts are necessary for the typeability of Buyer_1 to account for the fact that x is used *differently* in two distinct branches of the process. ⌟

▶ **Example 3.3** (parallel merge sort)**.** To illustrate an example of program that creates an unbounded number of sessions we model a parallel version of the merge sort algorithm.

$$\begin{aligned}
\mathit{Main} &\triangleq (s)(s[\mathsf{m}][\mathsf{w}]!\mathsf{req}.s[\mathsf{m}][\mathsf{w}]?\mathsf{res}.\mathsf{wait}\,s.\mathsf{done} \mid \mathit{Sort}\langle s[\mathsf{w}]\rangle)\\
\mathit{Sort}(x) &\triangleq x[\mathsf{m}]?\mathsf{req}.((t)(\mathit{Merge}\langle x, t[\mathsf{m}]\rangle \mid \mathit{Sort}\langle t[\mathsf{w}_1]\rangle \mid \mathit{Sort}\langle t[\mathsf{w}_2]\rangle) \oplus x[\mathsf{m}]!\mathsf{res}.\mathsf{close}\,x)\\
\mathit{Merge}(x,y) &\triangleq y[\mathsf{w}_1]!\mathsf{req}.y[\mathsf{w}_2]!\mathsf{req}.y[\mathsf{w}_1]?\mathsf{res}.y[\mathsf{w}_2]?\mathsf{res}.\mathsf{wait}\,y.x[\mathsf{m}]!\mathsf{res}.\mathsf{close}\,x
\end{aligned}$$

The program starts as a single session s in which a master m sends the initial collection of data to the worker w as a req message and waits for the result. The worker is modeled as a process *Sort* that decides whether to sort the data by itself (right branch of the choice in *Sort*), in which case it sends the result directly to the master, or to partition the collection (left branch of the choice in *Sort*). In the latter case, it creates a new session t in which it sends requests to two sub-workers w_1 and w_2, it gathers the partial results from them and gets back to the master with the complete result.

Since a worker may always choose to start two sub-workers in a new session, the number of sessions that may be created by this program is unbounded. At the same time, each worker may also choose to complete its task without creating new sessions. So, while in principle there exists a run of this program that keeps creating new sessions forever, this run is unfair according to Definition 2.2. ⌟

4 Multiparty Session Types and Fair Subtyping

In this section we define syntax and semantics of multiparty session types (Section 4.1) as well as an inference system for fair subtyping (Section 4.2).

4.1 Syntax and Semantics

A *session type* is a regular tree [16] coinductively generated by the productions below:

Session type $\quad S, T, U, V ::= \pi\mathsf{end} \mid \sum_{i\in I} \mathsf{p}\pi\mathsf{m}_i.S_i \mid \mathsf{p}\pi S.T$

The session type $\pi\mathsf{end}$ describes the behavior of a process that sends/receives a termination signal. The session type $\sum_{i\in I} \mathsf{p}\pi\mathsf{m}_i.S_i$ describes the behavior of a process that sends to or receives from the participant p one of the tags m_i and then behaves according to S_i. Note that the source or destination role p and the polarity π are the same in every branch. We require that I is not empty and $i, j \in I$ with $i \neq j$ implies $\mathsf{m}_i \neq \mathsf{m}_j$. Occasionally we write $\mathsf{p}\pi\mathsf{m}_1.S_1 + \cdots + \mathsf{p}\pi\mathsf{m}_n.S_n$ instead of $\sum_{i=1}^{n} \mathsf{p}\pi\mathsf{m}_i.S_i$. Finally, a session type $\mathsf{p}\pi S.T$ describes the behavior of a process that sends to or receives from the participant p an endpoint of type S and then behaves according to T. We often specify infinite session types as solutions of equations of the form $S = \cdots$ where the metavariable S may occur on the right hand side of $=$ guarded by at least one prefix. A regular tree satisfying such equation is guaranteed to exist and to be unique [16].

In order to describe a whole multiparty session at the level of types we introduce the notion of *session map*.

▶ **Definition 4.1** (session map). *A* session map *is a finite, partial map from roles to session types written* $\{\mathsf{p}_i \triangleright S_i\}_{i\in I}$. *We let M and N range over session maps, we write* $\mathsf{dom}(M)$ *for the domain of M, we write $M \mid N$ for the union of M and N when* $\mathsf{dom}(M) \cap \mathsf{dom}(N) = \emptyset$, *and we abbreviate the singleton map* $\{\mathsf{p} \triangleright S\}$ *as* $\mathsf{p} \triangleright S$.

We describe the evolution of a session at the level of types by means of a *labeled transition system* for session maps. Labels are generated by the grammar below:

Label $\quad \ell ::= \tau \mid \alpha \qquad$ **Action** $\quad \alpha, \beta ::= \pi\checkmark \mid \mathsf{p} \triangleright \mathsf{q}\pi\mathsf{m} \mid \mathsf{p} \triangleright \mathsf{q}\pi S$

The label τ represents either an internal action performed by a participant independently of the others or a synchronization between two participants. The labels of the form $\pi\checkmark$ describe the input/output of termination signals, whereas the labels of the form $\mathsf{p} \triangleright \mathsf{q}\pi\mathsf{m}$ and $\mathsf{p} \triangleright \mathsf{q}\pi S$ represent the input/output of a tag m or of an endpoint of type S.

The labeled transition system is defined by the rules in Table 4, most of which are straightforward. Rule [L-PICK] models the fact that the participant p may internally choose one particular tag m_k before sending it to q. The chosen tag is not negotiable with the receiver. Rule [L-TERMINATE] models termination of a session. A session terminates when there is exactly one participant waiting for the termination signal and all the others are sending it. This property follows from a straightforward induction on the derivation of $M \xrightarrow{?\checkmark} N$ using [L-TERMINATE] and [L-END]. The existence of a single participant waiting for the termination signal ensures that there is a uniquely determined continuation process after the session has been closed. Finally, rule [L-SYNC] models the synchronization between two participants performing complementary actions. The complement of an action α, denoted by $\overline{\alpha}$, is the partial operation defined by the equations

$$\overline{\mathsf{p} \triangleright \mathsf{q}\pi\mathsf{m}} \overset{\mathsf{def}}{=} \mathsf{q} \triangleright \mathsf{p}\overline{\pi}\mathsf{m} \qquad \overline{\mathsf{p} \triangleright \mathsf{q}\pi S} \overset{\mathsf{def}}{=} \mathsf{q} \triangleright \mathsf{p}\overline{\pi}S$$

Table 4 Labeled transition system for session maps.

$$\text{[L-END]}\ \frac{}{\mathsf{p} \triangleright \pi\mathsf{end} \xrightarrow{\pi\checkmark} \mathsf{p} \triangleright \pi\mathsf{end}} \qquad \text{[L-CHANNEL]}\ \frac{}{\mathsf{p} \triangleright \mathsf{q}\pi U.S \xrightarrow{\mathsf{p}\triangleright\mathsf{q}\pi U} \mathsf{p} \triangleright S}$$

$$\text{[L-PICK]}\ \frac{}{\mathsf{p} \triangleright \sum_{i\in I} \mathsf{q}!\mathsf{m}_i.S_i \xrightarrow{\tau} \mathsf{p} \triangleright \mathsf{q}!\mathsf{m}_k.S_k}\ k \in I \qquad \text{[L-TAG]}\ \frac{}{\mathsf{p} \triangleright \sum_{i\in I} \mathsf{q}\pi\mathsf{m}_i.S_i \xrightarrow{\mathsf{p}\triangleright\mathsf{q}\pi\mathsf{m}_k} \mathsf{p} \triangleright S_k}\ k \in I$$

$$\text{[L-TAU]}\ \frac{M \xrightarrow{\tau} M'}{M \mid N \xrightarrow{\tau} M' \mid N} \qquad \text{[L-TERMINATE]}\ \frac{M \xrightarrow{?\checkmark} M' \qquad N \xrightarrow{!\checkmark} N'}{M \mid N \xrightarrow{?\checkmark} M' \mid N'} \qquad \text{[L-SYNC]}\ \frac{M \xrightarrow{\overline{\alpha}} M' \qquad N \xrightarrow{\alpha} N'}{M \mid N \xrightarrow{\tau} M' \mid N'}$$

where $\overline{\pi}$ denotes the complement of the polarity π. The complement of actions of the form $\pi\checkmark$ is undefined, so rule [L-SYNC] cannot be applied to terminated sessions. Hereafter we write $\Rightarrow$ for the reflexive, transitive closure of $\xrightarrow{\tau}$ and $\xRightarrow{\alpha}$ for the composition $\Rightarrow\xrightarrow{\alpha}$.

We call *coherence* the property of multiparty sessions that we wish to enforce with our type system, namely the fact that a session can always terminate no matter how it evolves. We formulate coherence directly on the transition system of session maps, in line with the approach of Scalas and Yoshida [46] and without introducing global types.

▶ **Definition 4.2.** *We say that M is* coherent*, notation $\#M$, if $M \Rightarrow N$ implies $N \xRightarrow{?\checkmark}$.*

The term "coherence" is borrowed from Carbone et al. [8, 9], although the property is actually stronger than the one of Carbone et al. as it entails fair termination of multiparty sessions through Theorem 2.5. In particular, if we consider the reduction system whose states are session maps and whose reduction relation is $\xrightarrow{\tau}$, then $\#M$ implies M fairly terminating.

▶ **Example 4.3** (buyer-seller-carrier session map). Consider the session types

$$\begin{aligned} S_b &= \mathsf{seller}!\mathsf{add}.\mathsf{seller}!\mathsf{add}.S_b + \mathsf{seller}!\mathsf{pay}.!\mathsf{end} \\ S_s &= \mathsf{buyer}?\mathsf{add}.S_s + \mathsf{buyer}?\mathsf{pay}.\mathsf{carrier}!\mathsf{ship}.!\mathsf{end} \\ S_c &= \mathsf{seller}?\mathsf{ship}.?\mathsf{end} \end{aligned}$$

which describe the behavior of the processes *Buyer*, *Seller* and *Carrier* in Example 3.1. The session map $\mathsf{buyer} \triangleright S_b \mid \mathsf{seller} \triangleright S_s \mid \mathsf{carrier} \triangleright S_c$ is coherent. To see that, consider any interaction between the buyer and the seller. One of two cases applies: either the buyer has sent an even number of add messages to the seller, in which case it can send pay and the session eventually terminates, or the buyer has sent an odd number of add messages to the seller, in which case it can send one more add message followed by a pay message and once again the session eventually terminates. ⌟

Coherence allows us to provide a semantic definition of *fair subtyping*, the relation that defines the safe substitution principle for session endpoints in our type system.

▶ **Definition 4.4** (fair subtyping). *We say that S is a* fair subtype *of T, notation $S \sqsubseteq T$, if $M \mid \mathsf{p} \triangleright S$ coherent implies $M \mid \mathsf{p} \triangleright T$ coherent for every M and p.*

Definition 4.4 does not say much about the properties of fair subtyping except for the fact that it is a coherence-preserving preorder. For this reason, we devote Section 4.2 to defining an alternative characterization of fair subtyping that highlights its relationship with the standard subtyping relation for session types [23].

Table 5 Inference system for fair subtyping.

$$\text{[F-END]}\ \frac{}{\pi\mathsf{end} \leqslant_n \pi\mathsf{end}} \qquad \text{[F-CHANNEL]}\ \frac{S \leqslant_n T}{\mathsf{p}\pi U.S \leqslant_n \mathsf{p}\pi U.T} \qquad \text{[F-TAG-IN]}\ \frac{\forall i \in I : S_i \leqslant_{n_i} T_i \qquad \forall i \in I : n_i \leq n}{\sum_{i\in I} \mathsf{p}?\mathsf{m}_i.S_i \leqslant_n \sum_{i\in I\cup J} \mathsf{p}?\mathsf{m}_i.T_i}$$

$$\text{[F-TAG-OUT-1]}\ \frac{\forall i \in I : S_i \leqslant_{n_i} T_i \qquad \forall i \in I : n_i \leq n}{\sum_{i\in I} \mathsf{p}!\mathsf{m}_i.S_i \leqslant_n \sum_{i\in I} \mathsf{p}!\mathsf{m}_i.T_i} \qquad \text{[F-TAG-OUT-2]}\ \frac{\forall i \in I : S_i \leqslant_{n_i} T_i \qquad \exists i \in I : n_i < n}{\sum_{i\in I\cup J} \mathsf{p}!\mathsf{m}_i.S_i \leqslant_n \sum_{i\in I} \mathsf{p}!\mathsf{m}_i.T_i}$$

4.2 Inference System for Fair Subtyping

Consider the relation $\leqslant_n$ coinductively defined by the inference system in Table 5, where n ranges over natural numbers. The characterization of fair subtyping that we consider is the relation $\leqslant \stackrel{\text{def}}{=} \bigcup_{n\in\mathbb{N}} \leqslant_n$. The rules for deriving $S \leqslant_n T$ are quite similar to those of the standard subtyping relation for session types [23]: [F-END] states reflexivity of subtyping on terminated session types; [F-CHANNEL] relates higher-order session types with the same polarity and payload type; [F-TAG-IN] is the usual covariant rule for the input of tags (the set of tags in the larger session type includes those in the smaller one); [F-TAG-OUT-2] is the usual contravariant rule for the output of tags (the set of tags in the smaller session type includes those in the larger one). Overall, these rules entail a "simulation" between the behaviors described by S and T whereby all inputs offered by S are also offered by T and all outputs performed by T are also performed by S. The main differences between $\leqslant$ and the subtyping relation of Gay and Hole [23] are the presence of an invariant rule for outputs [F-TAG-OUT-1] and the natural number n annotating each subtyping judgment $S \leqslant_n T$. Intuitively, this number estimates how much S and T differ in terms of performed outputs. In all rules but [F-TAG-OUT-2], the annotation in the conclusion of the rule is just an upper bound of the annotations found in the premises. In [F-TAG-OUT-2], where the sets of output tags in related session types may differ, the annotation n is required to be a *strict* upper bound for at least one of the premises. That is, there must be at least one premise in which the annotation strictly decreases, while no restriction is imposed on the others. Intuitively, this ensures the existence of a tag shared by the two related session types whose corresponding continuations are slightly less different. So, the annotation n provides an upper bound to the number of applications of [F-TAG-OUT-2] along any path (i.e. any sequence of actions) shared by S and T that leads to termination. In the particular case when $n = 0$, the rule [F-TAG-OUT-2] cannot be applied, so that T may perform all the outputs also performed by S.

▶ **Example 4.5.** Consider the session type $S = \mathsf{seller}!\mathsf{add}.S + \mathsf{seller}!\mathsf{pay}.!\mathsf{end}$, which describes the behavior of the buyer in Equation (1) purchasing an arbitrary number of items, $T = \mathsf{seller}!\mathsf{add}.\mathsf{seller}!\mathsf{add}.T + \mathsf{seller}!\mathsf{pay}.!\mathsf{end}$, which describes the behavior of the buyer in Example 3.1 always purchasing an even number of items, and $U = \mathsf{seller}!\mathsf{add}.U$, which describes the behavior of a buyer attempting to purchase an infinite number of items without ever paying the seller. We have $S \leqslant T$ and $S \not\leqslant U$. Indeed, we can derive

$$\frac{\dfrac{\dfrac{\vdots}{S \leqslant_1 T}}{S \leqslant_2 \mathsf{seller}!\mathsf{add}.T}\ \text{[F-TAG-OUT-2]} \qquad \dfrac{}{!\mathsf{end} \leqslant_0 !\mathsf{end}}\ \text{[F-END]}}{S \leqslant_1 T}\ \text{[F-TAG-OUT-2]}$$

but there is no derivation for $S \leqslant_n U$ no matter how large n is chosen. Note that there are infinitely many sequences of actions of S that cannot be performed by both T and U. In particular, T cannot perform any sequence of actions consisting of an odd number of add outputs followed by a pay output, whereas U cannot perform any sequence of add outputs followed by a pay output. Nonetheless, there is a path shared by S and T that leads into a region of S and T in which no more differences are detectable. The annotations in the derivation tree measures the distance of each judgment from such region. In the case of S and U, there is no shared path that leads to a region where no differences are detectable. ⌟

▶ **Example 4.6.** Consider the session types $S = \mathsf{player}?\mathsf{play}.(\mathsf{player}!\mathsf{win}.S + \mathsf{player}!\mathsf{lose}.S) + \mathsf{player}?\mathsf{quit}.!\mathsf{end}$ and $T = \mathsf{player}?\mathsf{play}.\mathsf{player}!\mathsf{lose}.T + \mathsf{player}?\mathsf{quit}.!\mathsf{end}$ describing the behavior of two slot machines, an unbiased one in which the player may win at every play and a biased one in which the player never wins. If we try to build a derivation for $S \leqslant_n T$ we obtain

$$\dfrac{\dfrac{\dfrac{\vdots}{S \leqslant_{n-1} T}}{\mathsf{player}!\mathsf{win}.S + \mathsf{player}!\mathsf{lose}.S \leqslant_n \mathsf{player}!\mathsf{lose}.T}\,[\textsc{f-tag-out-1}] \qquad \dfrac{}{!\mathsf{end} \leqslant_n !\mathsf{end}}\,[\textsc{f-end}]}{S \leqslant_n T}\,[\textsc{f-tag-in}]$$

which would contain an infinite branch with strictly decreasing annotations. Therefore, we have $S \not\leqslant T$. In this case there exists a shared path leading into a region of S and T in which no more differences are detectable between the two protocols, but this path starts from an input. The fact that S is *not* a fair subtype of T has a semantic justification. Think of a player that deliberately insists on playing until it wins. This is possible when player interacts with the unbiased slot machine S but not with the biased one T. ⌟

In the rest of this section we study the fundamental properties of $\leqslant$, starting from the non-obvious fact that it is a preorder.

▶ **Theorem 4.7.** *$\leqslant$ is a preorder.*

While reflexivity of $\leqslant$ is trivial to prove (since [f-tag-out-2] is never necessary, it suffices to only consider judgments with a 0 annotation), transitivity is surprisingly complex. The challenging part of proving that from $S \leqslant_m U$ and $U \leqslant_n T$ we can derive $S \leqslant_k T$ is to come up with a feasible annotation k. As it turns out, such k depends not only on m and n, but also on annotations found in different regions of the derivation trees that prove $S \leqslant_m U$ and $U \leqslant_n T$. In particular, the "difference" of S and T is not simply the "maximum difference" or "the sum of the differences" of S and U and of U and T. More in detail, we first show that we can always find a derivation of $S \leqslant_m U$ where the rank annotations of all judgements occurring in it are below some $h \geq m$; then, the judgement $S \leqslant_k T$ is provable for $k = m + (1 + h)n$. For previous characterizations of fair subtyping [40, 42, 13, 14], transitivity has been established indirectly by relating the inference system of fair subtyping (Table 5) with its semantic definition (Definition 4.4). For Theorem 4.7 we are able to provide a direct proof [10].

Now we establish the connection between $\leqslant$ and $\sqsubseteq$ (Definition 4.4). First of all, we prove that $\leqslant$ is coherence-preserving just like $\sqsubseteq$ is.

▶ **Theorem 4.8** (soundness). *If $S \leqslant T$ then $S \sqsubseteq T$.*

The proof of this result relies on a key property of $\leqslant$ not enjoyed by the usual subtyping relation on session types [23]: when $S \leqslant T$ and $M \mid \mathsf{p} \triangleright S$ is coherent, the session map $M \mid \mathsf{p} \triangleright T$ can successfully terminate. The rank annotation on subtyping judgements is used to set up an appropriate inductive argument for proving this property.

Theorem 4.8 alone suffices to justify the adoption of $\leqslant$ as fair subtyping relation, but we are interested in understanding to which extent $\leqslant$ covers $\sqsubseteq$. In this respect, it is quite easy to see that there exist session types that are related by $\sqsubseteq$ but not by $\leqslant$. For example, consider $S = \mathsf{p}!\mathsf{a}.S$ and $T = \mathsf{p}?\mathsf{b}.T$ and observe that these two session types describe completely different protocols (the output of infinitely many a's in the case of S and the input of infinitely many b's in the case of T). In particular, we have $S \not\leqslant T$ and $T \not\leqslant S$ but also $S \sqsubseteq T$ and $T \sqsubseteq S$. That is, S and T are *unrelated* according to $\leqslant$ but they are *equivalent* according to $\sqsubseteq$. This equivalence is justified by the fact that there exists no coherent session map in which S and T could play any role, because none of them can ever terminate.

This discussion hints at the possibility that, if we restrict the attention to those session types that *can* terminate, which are the interesting ones as far as this work is concerned, then we can establish a tighter correspondence between $\leqslant$ and $\sqsubseteq$. We call such session types *bounded*, because they describe protocols for which termination is always within reach.

▶ **Definition 4.9** (bounded session type). *We say that a session type is* bounded *if all of its subtrees contain a* $\pi\mathsf{end}$ *leaf.*

Note that a *finite* session type is always bounded but not every bounded session type is finite. If we consider the reduction system in which states are session types and we have $S \to T$ if T is an immediate subtree of S, then S is bounded if and only if S is fairly terminating. Now, for the family of bounded session types we can prove a *relative completeness* result for $\leqslant$ with respect to $\sqsubseteq$.

▶ **Theorem 4.10** (relative completeness). *If S is bounded and $S \sqsubseteq T$ then $S \leqslant T$.*

The proof of Theorem 4.10 is done by contradiction. We show that, for any bounded S, if $S \leqslant T$ does not hold then we can build a session map M called *discriminator* such that $M \mid \mathsf{p} \triangleright S$ is coherent and $M \mid \mathsf{p} \triangleright T$ is not, which contraddicts the hypothesis $S \sqsubseteq T$. The boundedness of S is necessary to make sure that it is always possible to find a session map N such that $N \mid \mathsf{p} \triangleright S$ is coherent.

5 Type System

In this section we describe the type system for the calculus of multiparty sessions of Section 3. The typing judgments have the form $\Gamma \vdash^n P$, meaning that the process P is well typed in the typing context Γ and has rank n. As usual, the *typing context* is a map associating channels with session types and is meant to contain an association for each name in $\mathsf{fn}(P)$. We write $u_1 : S_1, \dots, u_n : S_n$ for the map with domain $\{u_1, \dots, u_n\}$ that associates u_i with S_i. Occtationally we write $\overline{u : S}$ for the same context, when the number and the specific associations are unimportant. We also assume that endpoints occurring in a typing context have different session names. That is, $s[\mathsf{p}], s[\mathsf{q}] \in \mathsf{dom}(\Gamma)$ implies $\mathsf{p} = \mathsf{q}$. This constraint makes sure that each well-typed process plays exactly one role in each of the sessions in which it participates. It is also a common assumption made in all multiparty session calculi. We use Γ and Δ to range over typing contexts, we write $\emptyset$ for the empty context and Γ, Δ for the union of Γ and Δ when they have disjoint domains and disjoint sets of session names. The *rank* n in a typing judgment estimates the number of sessions that P has to create and the number of casts that P has to perform in order to terminate. The fact that the rank is finite suggests that so is the effort required by P to terminate.

The typing rules are shown in Table 6 as a *generalized inference system* [3, 17, 11, 18] in which, roughly speaking, the singly-lined rules are interpreted coinductively and the doubly-lined rules – called *corules* – are interpreted inductively. We will come back with a more

Table 6 Typing rules.

$$
\textsc{[t-done]}\quad \begin{array}{c} \\ \hline \emptyset \vdash^n \mathsf{done} \end{array}
\qquad
\textsc{[t-call]}\quad \begin{array}{c} \overline{u : S} \vdash^n P\{\overline{u}/\overline{x}\} \\ \hline\hline \overline{u : S} \vdash^{n+m} A\langle\overline{u}\rangle \end{array}\ A : [\overline{S}; n], A(\overline{x}) \triangleq P
\qquad
\textsc{[t-wait]}\quad \begin{array}{c} \Gamma \vdash^n P \\ \hline \Gamma, u : ?\mathsf{end} \vdash^n \mathsf{wait}\, u.P \end{array}
$$

$$
\textsc{[t-close]}\quad \begin{array}{c} \\ \hline u : !\mathsf{end} \vdash^n \mathsf{close}\, u \end{array}
\qquad
\textsc{[t-channel-in]}\quad \begin{array}{c} \Gamma, u : T, x : S \vdash^n P \\ \hline \Gamma, u : \mathsf{p}?S.T \vdash^n u[\mathsf{p}]?(x).P \end{array}
\qquad
\textsc{[t-channel-out]}\quad \begin{array}{c} \Gamma, u : T \vdash^n P \\ \hline \Gamma, u : \mathsf{p}!S.T, v : S \vdash^n u[\mathsf{p}]!v.P \end{array}
$$

$$
\textsc{[t-tag]}\quad \begin{array}{c} \forall i \in I : \Gamma, u : S_i \vdash^n P_i \\ \hline \Gamma, u : \sum_{i\in I} \mathsf{p}\pi\mathsf{m}_i.S_i \vdash^n u[\mathsf{p}]\pi\{\mathsf{m}_i.P_i\}_{i\in I} \end{array}
\qquad
\textsc{[t-choice]}\quad \begin{array}{c} \Gamma \vdash^{n_1} P_1 \qquad \Gamma \vdash^{n_2} P_2 \\ \hline \Gamma \vdash^{n_k} P_1 \oplus P_2 \end{array}\ k \in \{1, 2\}
$$

$$
\textsc{[t-cast]}\quad \begin{array}{c} \Gamma, u : T \vdash^n P \\ \hline \Gamma, u : S \vdash^{m+n} \lceil u \rceil P \end{array}\ S \leqslant_m T
\qquad
\textsc{[t-par]}\quad \begin{array}{c} \forall i \in \{1, \dots, h\} : \Gamma_i, s[\mathsf{p}_i] : S_i \vdash^{n_i} P_i \\ \hline \Gamma_1, \dots, \Gamma_h \vdash^{1+n_1+\cdots+n_h} (s)(P_1 \mid \cdots \mid P_h) \end{array}\ \#\{\mathsf{p}_i \triangleright S_i\}_{i=1..h}
$$

$$
\textsc{[co-tag]}\quad \begin{array}{c} \Gamma, u : S_k \vdash^n P_k \\ \hline\hline \Gamma, u : \sum_{i\in I} \mathsf{p}\pi\mathsf{m}_i.S_i \vdash^n u[\mathsf{p}]\pi\{\mathsf{m}_i.P_i\}_{i\in I} \end{array}\ k \in I
\qquad
\textsc{[co-choice]}\quad \begin{array}{c} \Gamma \vdash^n P_k \\ \hline\hline \Gamma \vdash^n P_1 \oplus P_2 \end{array}\ k \in \{1, 2\}
$$

detailed intuition later on (Definition 5.1), although we will not provide a formal definition of the interpretation of a generalized inference system in this paper. The interested reader may refer to the cited literature for details. We type check a program $\{A_i(\overline{x_i}) \triangleq P_i\}_{i\in I}$ under a global set of assignments $\{A_i : [\overline{S_i}; n_i]\}_{i\in I}$ associating each process name A_i with a tuple of session types $\overline{S_i}$, one for each of the variables in $\overline{x_i}$, and a rank n_i. The program is well typed if $\overline{x_i : S_i} \vdash^{n_i} P_i$ is derivable for every $i \in I$, establishing that the tuple $\overline{S_i}$ corresponds to the way the variables $\overline{x_i}$ are used by P_i and that n_i is a feasible rank annotation for P_i. We now describe the typing rules in detail.

The rule [T-DONE] states that the terminated process is well typed in the empty context, to make sure that no unused channels are left behind. Note that done can be given any rank, since it performs no casts and it creates no new sessions. The rule [T-CALL] checks that a process invocation $A\langle\overline{u}\rangle$ is well typed by unfolding A into the process associated with A. The types associated with $\overline{u}$ must match those of the global assignment $A : [\overline{S}; n]$ and the rank of the process must be no greater than that of the invocation. The potential mismatch between the two ranks improves typeability in some corner cases. The rules [T-WAIT] and [T-CLOSE] concern processes that exchange termination signals. The channel being closed is consumed and, in the case of [T-WAIT], no longer available in the continuation P. Again, $\mathsf{close}\, u$ can be typed with any rank whereas the rank of $\mathsf{wait}\, u.P$ coincides with that of P. The rules [T-CHANNEL-IN] and [T-CHANNEL-OUT] deal with the exchange of channels in a quite standard way. Note that the actual type of the exchanged channel is required to coincide with the expected one. In particular, no covariance or contravariance of input and output respectively is allowed. Relaxing the typing rule in this way would introduce implicit applications of subtyping that may compromise fair termination [14]. In our type system, each application of subtyping must be explicitly accounted for as we will see when discussing [T-CAST]. Rule

[T-TAG] deals with the exchange of tags. Channels that are not used for such communication must be used in the same way in all branches, whereas the type of the channel on which the message is exchanged changes accordingly. All branches are required to have the same rank, which also corresponds to the rank of the process. Unlike other presentations of this typing rule [23], we require the branches in the process to be matched exactly by those in the type. Again, this is to avoid implicit application of subtyping, which might jeopardize fair termination. The rule [T-CHOICE] deals with non-deterministic choices and requires both continuations to be well typed in the same typing context. The judgment in the conclusion inherits the rank of one of the processes, typically the one with minimum rank. As we will see in Example 6.3, this makes it possible to model finite-rank processes that may create an unbounded number of sessions or that perform an unbounded number of casts.

The rule [T-CAST] models the substitution principle induced by fair subtyping: when $S \leqslant_m T$, a channel of type S can be used where a channel of type T is expected or, in dual fashion [22], a process using u according to T can be used in place of a process using u according to S. To keep track of this cast, the rank in the conclusion is augmented by the weight m of the subtyping relation between S and T. Note that the typing rule guesses the target type of the cast.

Finally, the rule [T-PAR] deals with session creation and parallel composition. This rule is inspired to the *multiparty cut* rule found in linear logic interpretations of multiparty session types [8, 9] and provides a straightforward way for enforcing deadlock freedom. Each process in the composition must be well typed in a slice of the typing context augmented with the endpoint corresponding to its role. The session map of the new session must be coherent, implying that it fairly terminates. The rank of the composition is one plus the aggregated rank of the composed processes, to account for the fact that one more session has been created. Recall that coherence is a property expressed on the LTS of session maps (Definition 4.2) in line with the approach of Scalas and Yoshida [46].

The typing rules described so far are interpreted *coinductively*. That is, in order for a rank n process P to be well typed in Γ there must be a *possibly infinite* derivation tree built with these rules and whose conclusion is the judgment $\Gamma \vdash^n P$. But in a generalized inference system like the one we are defining, this is not enough to establish that P is well typed. In addition, it must be possible to find *finite* derivation trees for all of the judgments occurring in this possibly infinite derivation tree using the discussed rules *and possibly* the corules, which we are about to describe. Since the additional derivation trees must be finite, all of their branches must end up with an application of [T-DONE] or [T-CLOSE], which are the only axioms in Table 6 corresponding to the only terminated processes in Table 1. So, the purpose of these finite typing derivations is to make sure that in every well-typed (sub-)process there exists a path that leads to termination. On the one hand, this is a sensible condition to require as our type system is meant to enforce fair process termination. On the other hand, insisting that these finite derivations can be built using only the typing rules discusses thus far is overly restrictive, for a process might have *one* path that leads to termination, but also alternative paths that lead to (recursive) process invocations. In fact, all of the processes we have discussed in Examples 3.1–3.3 are structured like this. The two corules [CO-CHOICE] and [CO-TAG] in Table 6 establish that, whenever a multi-branch process is dealt with, it suffices for *one* of the branches to lead to termination. A key detail to note in the case of [CO-CHOICE] is that the rank of the non-deterministic choice coincides with that of the branch that leads to termination. This makes sense recalling that the rank associated with a process represents the overall effort required for that process to terminate.

Let us recap the notion of well-typed process resulting from the typing rules of Table 6.

▶ **Definition 5.1** (well-typed process). *We say that P is* well typed *in the context Γ and has rank n if (1) there exists an arbitrary (possibly infinite) derivation tree obtained using the (singly-lined) rules in Table 6 and whose conclusion is $\Gamma \vdash^n P$ and (2) for each judgment in such tree there is a finite derivation obtained using the rules and the (doubly-lined) corules.*

▶ Remark 5.2. The term "*co*rule" seems to suggest that the rule should be *co*inductively interpreted. As we have seen above (Definition 5.1), corules are interpreted inductively. We have chosen to stick with the terminology used in the works that introduced generalized inference systems [3, 17]. ⌟

▶ **Example 5.3.** Let us show some typing derivations for fragments of Example 3.1 using the types S_b, S_s and S_c from Example 4.3. Concerning *Buyer*, we obtain the infinite derivation

$$\dfrac{\dfrac{\dfrac{\vdots}{x : S_b \vdash^0 \mathit{Buyer}\langle x\rangle}\,[\textsc{t-call}]}{x : \mathsf{seller}!\mathsf{add}.S_b \vdash^0 x[\mathsf{seller}]!\mathsf{add}.\mathit{Buyer}\langle x\rangle}\,[\textsc{t-tag}] \qquad \dfrac{}{x : !\mathsf{end} \vdash^0 \mathsf{close}\, x}\,[\textsc{t-close}]}{x : S_b \vdash^0 x[\mathsf{seller}]!\{\mathsf{add}.x[\mathsf{seller}]!\mathsf{add}.\mathit{Buyer}\langle x\rangle, \mathsf{pay}.\mathsf{close}\, x\}}\,[\textsc{t-tag}]$$

and, for each judgment in it, it is easy to find a finite derivation possibly using [CO-TAG]. Concerning *Main* we obtain

$$\dfrac{\dfrac{\vdots}{s[\mathsf{buyer}] : S_b \vdash^0 \mathit{Buyer}\langle s[\mathsf{buyer}]\rangle}\,[\textsc{t-call}] \quad \dfrac{\vdots}{s[\mathsf{seller}] : S_s \vdash^0 \mathit{Seller}\langle s[\mathsf{seller}]\rangle}\,[\textsc{t-call}] \quad \vdots}{\emptyset \vdash^1 (s)(\mathit{Buyer}\langle s[\mathsf{buyer}]\rangle \mid \mathit{Seller}\langle s[\mathsf{seller}]\rangle \mid \mathit{Carrier}\langle s[\mathsf{carrier}]\rangle)}\,[\textsc{t-par}]$$

where the application of [T-PAR] is justified by the fact that $\mathsf{buyer} \triangleright S_b \mid \mathsf{seller} \triangleright S_s \mid \mathsf{carrier} \triangleright S_c$ is coherent (Example 4.3). No participant creates new sessions or performs casts, so they all have zero rank. The rank of *Main* is 1 since it creates the session s. ⌟

We can prove a strong soundness result for our type system, stating that well-typed, closed processes can always successfully terminate no matter how they reduce.

▶ **Theorem 5.4** (soundness). *If $\emptyset \vdash^n P$ and $P \Rightarrow Q$, then $Q \Rightarrow\preccurlyeq \mathsf{done}$.*

There are several valuable implications of Theorem 5.4 on a well-typed, closed process P:

Deadlock freedom. If Q cannot reduce any further, then it must be (structurally precongruent to) done, namely there are no residual input/output actions.

Fair termination. Under the fairness assumption, Theorem 2.5 assures that P eventually reduces to done. This also implies that every session created by P eventually terminates.

Progress. If Q contains a sub-process with pending input/output actions, the fact that Q may reduce to done means that these actions are eventually performed.

The proof of Theorem 5.4 is essentially composed of a standard subject reduction result showing that typing is preserved by reductions and a proof that every well-typed process other than done may always reduce in such a way that a suitably defined *well-founded measure* strictly decreases. The measure is a lexicographically ordered pair of natural numbers with the following meaning: the first component measures the number of sessions that must be created and the total weight of casts that must be performed in order for the process to terminate (this information is essentially the rank we associate with typing judgments); the second component measures the overall effort required to terminate every session that has already been created (these sessions are identified by the fact that their restriction occurs unguarded in the process). We account for this effort by measuring the shortest reduction that terminates a coherent session map (Definition 4.2). The reason why we need two quantities in the measure is that in general every application of fair subtyping may *increase* the length of the shortest reduction that terminates a coherent session map. So, when casts are performed the second component of the measure may increase, but the first component reduces. As a final remark, it should be noted that the overall measure associated with a well-typed process *may also increase*, for example if new sessions are created (Example 3.3). However, one particular reduction that decreases the measure is always guaranteed to exist.

We conclude this section discussing a few more examples that motivate the features of the type system that are key for ensuring fair program termination.

▶ **Example 5.5.** To see simple examples of processes whose ill/well typing crucially depends on the fact that we use a generalized inference system consider the definitions

$$A \triangleq A \qquad B \triangleq B \oplus B \qquad C \triangleq C \oplus \mathsf{done}$$

which define a stuck process A, a diverging process B and a fairly terminating process C that admits an infinite reduction. For them we can find the infinite typing derivations below:

$$\dfrac{\dfrac{\vdots}{\emptyset \vdash^0 A}\,[\textsc{t-call}]}{\emptyset \vdash^0 A}\,[\textsc{t-call}] \qquad \dfrac{\dfrac{\vdots \qquad \dfrac{\vdots}{\emptyset \vdash^0 B}\,[\textsc{t-call}]}{\emptyset \vdash^0 B \oplus B}\,[\textsc{t-choice}]}{\emptyset \vdash^0 B}\,[\textsc{t-call}] \qquad \dfrac{\dfrac{\dfrac{\vdots}{\emptyset \vdash^0 C}\,[\textsc{t-call}] \qquad \dfrac{}{\emptyset \vdash^0 \mathsf{done}}\,[\textsc{t-done}]}{\emptyset \vdash^0 C \oplus \mathsf{done}}\,[\textsc{t-choice}]}{\emptyset \vdash^0 C}$$

However, only for C it is possible to find a finite typing derivation using the corule [CO-CHOICE]. So, A and B are ill typed, whereas C is well typed. This is consistent with the fact that only C can always reduce to the successfully terminated process done. ⌟

▶ **Example 5.6** (infinitely ranked processes)**.** The mere existence of a path that leads to termination ensured by the generalized interpretation of the typing rules in Table 6 does not always guarantee that the process is actually able to terminate. An example where this is the case is shown by the process A defined as

$$A \triangleq (s)(s[\mathsf{p}][\mathsf{q}]!\{\mathsf{a}.\mathsf{close}\, s[\mathsf{p}], \mathsf{b}.\mathsf{wait}\, s[\mathsf{p}].A\} \mid s[\mathsf{q}][\mathsf{p}]?\{\mathsf{a}.\mathsf{wait}\, s[\mathsf{q}].A, \mathsf{b}.\mathsf{close}\, s[\mathsf{q}]\})$$

which creates a session s and splits as two parallel sub-processes connected by s. Each sub-process has a path that leads to termination but, because of the way they synchronize, when one sub-process terminates the other one restarts A. For A it would be possible to build a finite typing derivation with the help of [CO-TAG], but A is ill typed because it cannot be assigned a finite rank, since it creates a new session at each recursive invocation.

Further examples of infinitely ranked processes, including ones where the rank is affected by the presence of casts, are discussed by Ciccone and Padovani [14] for binary sessions and can be easily reframed in our multiparty setting. ⌟

6 Advanced Examples

▶ **Example 6.1.** In this example we show that the process $Buyer_1$ playing the role b_1 in the inner session of Example 3.2 is well typed. For clarity, we recall its definition here:

$$\begin{aligned} Buyer_1(x, y) \triangleq y[\mathsf{b}_2]!\{&\mathsf{split}.y[\mathsf{b}_2]?\{\mathsf{yes}.\lceil x\rceil x[\mathsf{s}]!\mathsf{ok}.x[\mathsf{c}]?\mathsf{box}.\mathsf{wait}\, x.\mathsf{wait}\, y.\mathsf{done}, \\ &\qquad\qquad\qquad\qquad \mathsf{no}.Buyer_1\langle x, y\rangle\}, \\ &\mathsf{giveup}.\mathsf{wait}\, y.\lceil x\rceil x[\mathsf{s}]!\mathsf{cancel}.\mathsf{wait}\, x.\mathsf{done}\} \end{aligned}$$

We wish to build a typing derivation showing that Buyer_1 has rank 1 and uses x and y respectively according to S and T, where $S = \mathsf{s}!\mathsf{ok}.\mathsf{c}?\mathsf{box}.?\mathsf{end} + \mathsf{s}!\mathsf{cancel}.?\mathsf{end}$ and $T = \mathsf{b}_2!\mathsf{split}.(\mathsf{b}_2?\mathsf{yes}.?\mathsf{end} + \mathsf{b}_2?\mathsf{no}.T) + \mathsf{b}_2!\mathsf{giveup}.?\mathsf{end}$. As it has been noted previously, what makes this process interesting is that it uses the endpoint x differently depending on the messages it exchanges with b_2 on y. Since rule [T-TAG] requires any endpoint other than the one on which messages are exchanged to have the same type, the only way Buyer_2 can be declared well typed is by means of the casts that occur in its body. For the branch in which Buyer_1 proposes to split the payment we obtain the following derivation tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash^0 \mathsf{done}}\,[\text{T-DONE}]}{y : ?\mathsf{end} \vdash^0 \mathsf{wait}\, y.\mathsf{done}}\,[\text{T-WAIT}]}{x : ?\mathsf{end}, y : ?\mathsf{end} \vdash^0 \mathsf{wait}\, x \dots}\,[\text{T-WAIT}]}{x : \mathsf{c}?\mathsf{box}.?\mathsf{end}, y : ?\mathsf{end} \vdash^0 x[\mathsf{c}]?\mathsf{box} \dots}\,[\text{T-TAG}]}{x : \mathsf{s}!\mathsf{ok}.\mathsf{c}?\mathsf{box}.?\mathsf{end}, y : ?\mathsf{end} \vdash^0 x[\mathsf{s}]!\mathsf{ok} \dots}\,[\text{T-TAG}]}{x : S, y : ?\mathsf{end} \vdash^1 \lceil x \rceil \cdots}\,[\text{T-CAST}] \qquad \dfrac{\vdots}{x : S, y : T \vdash^1 \mathit{Buyer}_1\langle x, y \rangle}\,[\text{T-CALL}]}{x : S, y : \mathsf{b}_2?\mathsf{yes}.?\mathsf{end} + \mathsf{b}_2?\mathsf{no}.T \vdash^1 y[\mathsf{b}_2]?\{\mathsf{yes} \dots, \mathsf{no} \dots\}}\,[\text{T-TAG}]$$

Note how the application of [T-CAST] is key to change the type of x in the branch where the proposed split is accepted by b_2. In that branch, x is deterministically used to send an ok message and we leverage on the fair subtyping relation $S \leqslant_1 \mathsf{s}!\mathsf{ok}.\mathsf{c}?\mathsf{box}.?\mathsf{end}$.

For the branch in which Buyer_1 sends giveup we obtain the following derivation tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash^0 \mathsf{done}}\,[\text{T-DONE}]}{x : ?\mathsf{end} \vdash^0 \mathsf{wait}\, x.\mathsf{done}}\,[\text{T-WAIT}]}{x : \mathsf{s}!\mathsf{cancel}.?\mathsf{end} \vdash^0 x[\mathsf{s}]!\mathsf{cancel}.\mathsf{wait}\, x.\mathsf{done}}}{x : S \vdash^1 \lceil x \rceil x[\mathsf{s}]!\mathsf{cancel}.\mathsf{wait}\, x.\mathsf{done}}\,[\text{T-CAST}]}{x : S, y : ?\mathsf{end} \vdash^1 \mathsf{wait}\, y.\lceil x \rceil x[\mathsf{s}]!\mathsf{cancel}.\mathsf{wait}\, x.\mathsf{done}}\,[\text{T-WAIT}]$$

Once again the cast is necessary to change the type of x, but this time leveraging on the fair subtyping relation $S \leqslant_1 \mathsf{s}!\mathsf{cancel}.?\mathsf{end}$. These two derivations can then be combined to complete the proof that the body of Buyer_1 is well typed:

$$\dfrac{\vdots \qquad\qquad \vdots}{x : S, y : T \vdash^1 y[\mathsf{b}_2]!\{\mathsf{split} \dots, \mathsf{giveup} \dots\}}\,[\text{T-TAG}]$$

Clearly, it is also necessary to find finite derivation trees for all of the judgments shown above. This can be easily achieved using the corule [CO-TAG]. ⌟

▶ **Example 6.2.** Casts can be useful to reconcile the types of a channel that is used differently in different branches of a non-deterministic choice. For example, below is an alternative modeling of *Buyer* from Example 3.1 where we abbreviate seller to s for convenience:

$$B(x) \triangleq \lceil x \rceil x[\mathsf{s}]!\mathsf{add}.x[\mathsf{s}]!\mathsf{add}.B\langle x \rangle \oplus \lceil x \rceil x[\mathsf{s}]!\mathsf{pay}.\mathsf{close}\, x$$

Note that x is used for sending two add messages in the left branch of the non-deterministic choice and for sending a single pay message in the right branch. Given the session type $S = \mathsf{s}!\mathsf{add}.S + \mathsf{s}!\mathsf{pay}.!\mathsf{end}$ and using the fair subtyping relations $S \leqslant_2 \mathsf{s}!\mathsf{add}.\mathsf{s}!\mathsf{add}.S$ and $S \leqslant_1 \mathsf{s}!\mathsf{pay}.!\mathsf{end}$ we can obtain the following typing derivation for the body of B:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{x : S \vdash^1 B\langle x\rangle}\,[\text{T-CALL}]}{x : \mathsf{s}!\mathsf{add}.S \vdash^1 x[\mathsf{s}]!\mathsf{add}.B\langle x\rangle}\,[\text{T-TAG}]}{x : \mathsf{s}!\mathsf{add}.\mathsf{s}!\mathsf{add}.S \vdash^1 x[\mathsf{s}]!\mathsf{add}.x[\mathsf{s}]!\mathsf{add}.B\langle x\rangle}\,[\text{T-TAG}]}{x : S \vdash^3 \lceil x\rceil x[\mathsf{s}]!\mathsf{add}.x[\mathsf{s}]!\mathsf{add}.B\langle x\rangle}\,[\text{T-CAST}] \qquad \dfrac{\dfrac{\dfrac{}{x : !\mathsf{end} \vdash^0 \mathsf{close}\, x}\,[\text{T-CLOSE}]}{x : \mathsf{s}!\mathsf{pay}.!\mathsf{end} \vdash^0 x[\mathsf{s}]!\mathsf{pay}.\mathsf{close}\, x}\,[\text{T-TAG}]}{x : S \vdash^1 \lceil x\rceil x[\mathsf{s}]!\mathsf{pay}.\mathsf{close}\, x}\,[\text{T-CAST}]$$

$$\overline{x : S \vdash^1 \lceil x\rceil x[\mathsf{s}]!\mathsf{add}.x[\mathsf{s}]!\mathsf{add}.B\langle x\rangle \oplus \lceil x\rceil x[\mathsf{s}]!\mathsf{pay}.\mathsf{close}\, x}\;[\text{T-CHOICE}]$$

In general, the transformation $u[\mathsf{p}]!\{\mathsf{m}_i.P_i\}_{i=1..n} \rightsquigarrow \lceil u\rceil u[\mathsf{p}]!\mathsf{m}_1.P_1 \oplus \cdots \oplus \lceil u\rceil u[\mathsf{p}]!\mathsf{m}_n.P_n$ does not always preserve typing, so it is not always possible to encode the output of tags using casts and non-deterministic choices. As an example, the definition

$$\mathit{Slot}(x) \triangleq x[\mathsf{player}]?\{\mathsf{play}.x[\mathsf{player}]!\{\mathsf{win}.\mathit{Slot}\langle x\rangle, \mathsf{lose}.\mathit{Slot}\langle x\rangle\}, \mathsf{quit}.\mathsf{close}\, x\}$$

implements the unbiased slot machine of Example 4.6. It is easy to see that *Slot* is well typed under the global type assignment $\mathit{Slot} : [T; 0]$ where $T = \mathsf{player}?\mathsf{play}.(\mathsf{player}!\mathsf{win}.T + \mathsf{player}!\mathsf{lose}.T) + \mathsf{player}?\mathsf{quit}.!\mathsf{end}$. In particular, *Slot* has rank 0 since it performs no casts and it creates no sessions. If we encode the tag output in *Slot* using casts and non-deterministic choices we end up with the following process definition, which is ill typed because it cannot be given a finite rank:

$$\mathit{Slot}(x) \triangleq x[\mathsf{player}]?\{\mathsf{play}.(\lceil x\rceil x[\mathsf{player}]!\mathsf{win}.\mathit{Slot}\langle x\rangle \oplus \lceil x\rceil x[\mathsf{player}]!\mathsf{lose}.\mathit{Slot}\langle x\rangle), \mathsf{quit}.\mathsf{close}\, x\}$$

The difference between this version of *Slot* and the above definition of B is that *Slot* always recurs after a cast, so it is not obvious that finitely many casts suffice in order for *Slot* to terminate. ⌟

▶ **Example 6.3.** Here we provide evidence that the process definitions in Example 3.3 are well typed, even if they model processes that can open arbitrarily many sessions. In that example, the most interesting process definition is that of the worker *Sort*, which is recursive and may create a new session. In contrast, *Merge* is finite and *Main* only refers to *Sort*. We claim that these process definitions are well typed under the global type assignments

$$\mathit{Main} : [(); 1] \qquad \mathit{Sort} : [U; 0] \qquad \mathit{Merge} : [T, V; 0]$$

where $T = \mathsf{m}!\mathsf{res}.!\mathsf{end}$, $U = \mathsf{m}?\mathsf{req}.T$ and $V = \mathsf{w}_1!\mathsf{req}.\mathsf{w}_2!\mathsf{req}.\mathsf{w}_1?\mathsf{res}.\mathsf{w}_2?\mathsf{res}.?\mathsf{end}$.

For the branch of *Sort* that creates a new session we obtain the derivation tree

$$\dfrac{\dfrac{\vdots}{x : T, t[\mathsf{m}] : V \vdash^0 \mathit{Merge}\langle x, t[\mathsf{m}]\rangle}\,[\text{T-CALL}] \qquad \dfrac{\vdots}{t[\mathsf{w}_i] : U \vdash^0 \mathit{Sort}\langle t[\mathsf{w}_i]\rangle}\,[\text{T-CALL}], i = 1, 2}{x : T \vdash^1 (t)(\mathit{Merge}\langle x, t[\mathsf{m}]\rangle \mid \mathit{Sort}\langle t[\mathsf{w}_1]\rangle \mid \mathit{Sort}\langle t[\mathsf{w}_2]\rangle)}\,[\text{T-PAR}]$$

where the rank 1 derives from the fact that the created session involves three zero-ranked participants. For the body of *Sort* we obtain the following derivation tree:

$$\dfrac{\dfrac{\dfrac{\vdots}{x : T \vdash^1 (t)(\mathit{Merge}\langle x, t[\mathsf{m}]\rangle \mid \mathit{Sort}\langle t[\mathsf{w}_1]\rangle \mid \cdots)}\,[\text{T-PAR}] \qquad \dfrac{\dfrac{}{x : !\mathsf{end} \vdash^0 \mathsf{close}\, x}\,[\text{T-CLOSE}]}{x : T \vdash^0 x[\mathsf{m}]!\mathsf{res}.\mathsf{close}\, x}\,[\text{T-TAG}]}{x : T \vdash^0 (t)(\mathit{Merge}\langle x, t[\mathsf{m}]\rangle \mid \mathit{Sort}\langle t[\mathsf{w}_1]\rangle \mid \cdots) \oplus x[\mathsf{m}]!\mathsf{res}.\mathsf{close}\, x}\,[\text{T-CHOICE}]}{x : U \vdash^0 x[\mathsf{m}]?\mathsf{req}.((t)(\mathit{Merge}\langle x, t[\mathsf{m}]\rangle \mid \mathit{Sort}\langle t[\mathsf{w}_1]\rangle \mid \cdots) \oplus x[\mathsf{m}]!\mathsf{res}.\mathsf{close}\, x)}\,[\text{T-TAG}]$$

In the application of the rule [T-CHOICE], the rank of the whole choice coincides with that of the branch in which no new sessions are created. This way we account for the fact that, even though *Sort* *may* create a new session, it does not *have to* do so in order to terminate. ⌟

7 Related Work

Fair termination of binary sessions. Our type system is both a refinement and an extension of the one presented by Ciccone and Padovani [14], which ensures the fair termination of *binary* sessions. The main elements of the two type systems are closely related, but there are some key differences. In that work, the fairness assumption being made is *strong fairness* [21, 4, 36, 47] which guarantees fair termination of binary sessions *at the level of types* but not necessarily *at the level of processes.* The key difference between types and processes is that types generate *finite-state* reduction systems (because of their regularity) whereas processes may generate *infinite-state* reduction systems. While strong fairness is known to be the strongest possible fairness assumption for finite-state systems [48], it is not strong enough to make the right-to-left direction of Theorem 2.5 hold for infinite-state systems. In fact, it can be shown that strong fairness and the fairness assumption we make in this work (Definition 2.2) are unrelated for infinite-state reduction systems, in the sense that there exist fair runs that are not strongly fair and there exist strongly fair runs that are not fair runs. The fairness assumption we make in this work is general enough so that it can be related to both types (Definition 4.2) and processes (Theorem 5.4) through Theorem 2.5. The main advantage of working with native multiparty sessions is that they enable the natural modeling of interactions involving multiple participants in possibly cyclic network topologies, like those in Examples 3.2 and 3.3. Another difference and contribution of our work compared to the one of Ciccone and Padovani [14] is that the definition of the fair subtyping relation is simpler. In particular, the inference system we provide (Table 5) does not make use of corules [13, 14] nor does it require auxiliary predicates [40, 42].

Liveness properties of multiparty sessions. The enforcement of liveness properties has always been a key aspect of session type systems, although previous works have almost exclusively focused on progress rather than on (fair) termination. Scalas and Yoshida [46] define a general framework for ensuring safety and liveness properties of multiparty sessions. In particular, they define a hierarchy of three liveness predicates to characterize “live” sessions that enjoy progress. They also point out that the coarsest liveness property in this hierarchy, which is the one more closely related to fair termination, cannot be enforced by their type system. In part, this is due to the fact that their type system relies on a standard subtyping relation for session types [23] instead of fair subtyping [40, 42]. As we have seen in Section 5, even for single-session programs the mere adoption of fair subtyping is not enough and it is necessary to meet additional requirements (Examples 5.5 and 5.6). The work of van Glabbeek et al. [48] presents a type system for multiparty sessions that ensures progress and is not only sound but also complete. The fairness assumption they make – called *justness* – is substantially weaker than our own (Definition 2.2) and such that the unfair runs are those in which some interactions between participants are systematically discriminated in favor of other interactions involving a disjoint set of independent participants. For this reason, their progress property is in between the two more restrictive liveness predicates of Scalas and Yoshida [46] and can only be guaranteed when it is independent of the behavior of the other participants of the same session. In the end, simple sessions like those described in Examples 3.1–3.3 fall outside the scope of these works as far as liveness properties are concerned.

Another major difference between our work and the ones cited above [46, 48] is that fair termination, unlike progress, enables compositional reasoning and so we are able to enforce a global liveness property (Theorem 5.4) *even in the presence of multiple sessions* (see Examples 3.2 and 3.3). Notable examples of multiparty session type systems ensuring progress also in the presence of multiple (possibly interleaved) sessions are provided by Padovani et al. [43] and by Coppo et al. [15]. This is achieved by a rich type structure that prevents mutual dependencies between different sessions. In any case, these works do not address sessions in which progress may depend on choices made by session participants.

Termination of binary sessions. Termination is a liveness property that can be guaranteed when finite session types are considered [44]. As soon as infinite session types are considered, many session type systems weaken the guaranteed property to deadlock freedom. Lindley and Morris [38] define a type system for a functional language with session primitives and recursive session types that is strongly normalizing. That is, a well-typed program along with all the sessions it creates is guaranteed to terminate. This strong result is due to the fact that the type language is equipped with least and greatest fixed point operators that are required to match each other by duality. Termination is strictly stronger than fair termination. In particular, there exist fairly terminating programs that are not terminating because they allow reductions of unbounded length (see Examples 3.1–3.3).

Liveness properties in the π-calculus. Kobayashi [29] defines a behavioral type system that guarantees lock freedom in the π-calculus. Lock freedom is a liveness property akin to progress for sessions, except that it applies to *any* communication channel (shared or private). Padovani [41] adapts and extends the type system of Kobayashi [29] to enforce lock freedom in the *linear π-calculus* [32], into which binary sessions can be encoded [19]. All of these works annotate types with numbers representing finite upper bounds to the number of interactions needed to unblock a particular input/output action. For this reason, none of our key examples (Examples 3.1–3.3) is in the scope of these analysis techniques. Kobayashi and Sangiorgi [33] show how to enforce lock freedom by combining deadlock freedom and termination. Our work can be seen as a generalization of this approach whereby we enforce lock freedom by combining deadlock freedom (through a mostly conventional session type system) and *fair* termination. Since fair termination is coarser than termination, the family of programs for which lock freedom can be proved is larger as well.

Deadlock freedom. Our type system enforces deadlock freedom essentially thanks to the shape of the rule [T-PAR] which is inspired to the cut rule of linear logic. This rule has been applied to session type systems for binary sessions [49, 6, 38] and subsequently extended to multiparty sessions [8, 9]. In the latter case, the rule – dubbed *multiparty cut* – requires a coherence condition among cut types establishing that the session types followed by the single participants adhere to a so-called global type describing the multiparty session as a whole. The rule [T-PAR] adopts with schema, except that the coherence condition is stronger to entail fair session termination. The key principle of these formulations of the cut rule as a typing rule for parallel processes is to impose a tree-like network topology, whereby two parallel processes can share at most one channel. In the multiparty case, cyclic network topologies can be modeled within each session (Example 3.3) since coherence implies deadlock freedom.

Having a single construct that merges session restriction and parallel composition allows for a simple formulation of the typing rules so that dealock freedom is easily guaranteed. However, many session calculi separate these two forms in line with the original presentation of the π-calculus. We think that our type system can be easily reformulated to support distinct session restriction and parallel composition by means of hypersequents [34, 35].

A more liberal version of the cut rule, named multi-cut and inspired to Gentzen's "mix" rule, is considered by Abramsky et al. [1] enabling processes to share more than one channel. In this setting, deadlock freedom is lost but can be recovered by means of a richer type structure that keeps track of the dependencies between different channels. This approach has been pioneered by Kobayashi [29, 30] for the π-calculus and later on refined by Padovani [41]. Other approaches to ensure deadlock freedom based on *dependency/connectivity graphs* that capture the network topology implemented by processes have been studied by Carbone and Debois [7], Kobayashi and Laneve [31], de'Liguoro and Padovani [20], and Jacobs et al. [28].

8 Concluding Remarks

Sessions ought to terminate. Until recently this property has been granted only for sessions whose duration is bounded. In this work we have presented the first type system ensuring the *fair termination* of multiparty sessions, that is a termination property under the assumption that, if termination is always reachable, then it is eventually achieved. Fair termination is stronger than weak termination but substantially weaker than strong normalization. In particular, fair termination does not rule out infinite runs of well-typed processes as long as they purposefully eschew termination. When fair termination is combined with the usual safety properties of sessions, it entails a strong progress property whereby *any* pending action is eventually performed. Our type system is the first ensuring such strong progress property for multiparty (and possibly multiple) sessions.

A cornerstone element of the type system is *fair subtyping*, a coherence-preserving refinement of the standard subtyping relation for session types [23]. In this work, we have also contributed a new characterization of fair subtyping (Table 5 and Theorems 4.8 and 4.10) that is substantially simpler than previous ones [40, 42, 13, 14] since it does not require auxiliary predicates nor the use of a generalized inference system [3, 17, 13, 14]. Thanks to this new characterization we have been able to prove the transitivity of fair subtyping (Theorem 4.7) without relying on its (relative) completeness with respect to its semantic counterpart (Definition 4.4).

The decidability of fair subtyping and of type checking follow from analogous results for binary sessions [14]. The rank of processes can be inferred using the same algorithm that works for the binary case [14, auxiliary material]. Considering that fair subtyping for multiparty session types coincides with fair subtyping for binary session types except for the presence of roles, it would be easy to adapt the type checking tool FairCheck [12] to the process language we consider in this paper. The most relevant difference would be the algorithm for deciding the coherence of a session map, which is somewhat more complex than that for the compatibility between two session types. As for the binary setting, to which extent the type system is amenable to full type reconstruction is yet to be established. In particular, a hypothetical type inference algorithm would have to be able to solve fair subtyping inequations and this problem has not been investigated yet. Another open question that may have a relevant practical impact is whether the type system remains *sound* in a setting where communications are *asynchronous*. We expect the answer to be positive, as is the case for other synchronous multiparty session types systems [46], but we have not worked out the details yet.

In this paper we have focused on the theoretical aspects of fairly terminating multiparty sessions. A natural development of this work is its application to a real programming environment. We envision two approaches that can be followed to this aim. A bottom-up approach may apply our static analysis technique to a program (in our process calculus) that is extracted from actual code and that captures the code's communication semantics. We expect that suitable annotations may be necessary to identify those branching parts of the code that represent non-deterministic choices in the program. Most typically, these branches will correspond to finite loops or to queries made to the human user of the program that have several different continuations. A top-down approach may provide programmers with a *generative tool* that, starting from a global specification in the form of a *global type* [26], produces template code that is "well-typed by design" and that the programmer subsequently instantiates to a specific application. Scribble [50, 2] is an example of such a tool. Interestingly, the usual notion of global type projectability is not sufficient to entail that the session map resulting from a projection is coherent. However, coherence would be guaranteed by requiring that the projected global type is fairly terminating.

Finally, we plan to investigate the adaptation of the type system for ensuring the fair termination in the popular actor-based model. This is a drastically different setting in which the order of messages is not as controllable as in the case of sessions. As a consequence, type based analyses require radically different formalisms such as mailbox types [20], for which the study of fair subtyping and of type systems enforcing fair termination is unexplored.

References

1 Samson Abramsky, Simon J. Gay, and Rajagopal Nagarajan. Interaction categories and the foundations of typed concurrent programming. In Manfred Broy, editor, *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*, pages 35–113, 1996.

2 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Found. Trends Program. Lang.*, 3(2-3):95–230, 2016. doi:10.1561/2500000031.

3 Davide Ancona, Francesco Dagnino, and Elena Zucca. Generalizing inference systems by coaxioms. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 29–55. Springer, 2017. doi:10.1007/978-3-662-54434-1_2.

4 Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 189–198. ACM Press, 1987. doi:10.1145/41625.41642.

5 Mario Bravetti and Gianluigi Zavattaro. A theory of contracts for strong service compliance. *Math. Struct. Comput. Sci.*, 19(3):601–638, 2009. doi:10.1017/S0960129509007658.

6 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Math. Struct. Comput. Sci.*, 26(3):367–423, 2016. doi:10.1017/S0960129514000218.

7 Marco Carbone and Søren Debois. A graphical approach to progress for structured communication in web services. In Simon Bliudze, Roberto Bruni, Davide Grohmann, and Alexandra Silva, editors, *Proceedings Third Interaction and Concurrency Experience: Guaranteed Interaction, ICE 2010, Amsterdam, The Netherlands, 10th of June 2010*, volume 38 of *EPTCS*, pages 13–27, 2010. doi:10.4204/EPTCS.38.4.

8 Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, volume 59 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CONCUR.2016.33.

9 Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. *Acta Informatica*, 54(3):243–269, 2017. doi:10.1007/s00236-016-0285-y.

10 Luca Ciccone, Francesco Dagnino, and Luca Padovani. Fair Termination of Multiparty Sessions. Technical report, Università di Torino and Università di Genova, 2022. URL: https://arxiv.org/abs/2205.08786.

11 Luca Ciccone, Francesco Dagnino, and Elena Zucca. Flexible coinduction in agda. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 13:1–13:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.13.

12 Luca Ciccone and Luca Padovani. FairCheck. GitHub repository, 2021. URL: https://github.com/boystrange/FairCheck.

13 Luca Ciccone and Luca Padovani. Inference Systems with Corules for Fair Subtyping and Liveness Properties of Binary Session Types. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP'21)*, volume 198 of *LIPIcs*, pages 125:1–125:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2021.125.

14 Luca Ciccone and Luca Padovani. Fair termination of binary sessions. *Proc. ACM Program. Lang.*, 6(POPL):1–30, 2022. doi:10.1145/3498666.

15 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Math. Struct. Comput. Sci.*, 26(2):238–302, 2016. doi:10.1017/S0960129514000188.

16 Bruno Courcelle. Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–169, 1983. doi:10.1016/0304-3975(83)90059-2.

17 Francesco Dagnino. Coaxioms: flexible coinductive definitions by inference systems. *Log. Methods Comput. Sci.*, 15(1), 2019. doi:10.23638/LMCS-15(1:26)2019.

18 Francesco Dagnino. *Flexible Coinduction.* PhD thesis, DIBRIS, University of Genoa, January 2021.

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

20 Ugo de'Liguoro and Luca Padovani. Mailbox types for unordered interactions. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPIcs*, pages 15:1–15:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ECOOP.2018.15.

21 Nissim Francez. *Fairness.* Texts and Monographs in Computer Science. Springer, 1986. doi:10.1007/978-1-4612-4886-6.

22 Simon J. Gay. Subtyping supports safe session substitution. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 95–108. Springer, 2016. doi:10.1007/978-3-319-30936-1_5.

23 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005. doi:10.1007/s00236-005-0177-z.

24 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

25 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.

26 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.

27 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.

28 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.*, 6(POPL):1–33, 2022. doi:10.1145/3498662.

29 Naoki Kobayashi. A type system for lock-free processes. *Inf. Comput.*, 177(2):122–159, 2002. doi:10.1006/inco.2002.3171.

30 Naoki Kobayashi. A new type system for deadlock-free processes. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2006. doi:10.1007/11817949_16.

31 Naoki Kobayashi and Cosimo Laneve. Deadlock analysis of unbounded process networks. *Inf. Comput.*, 252:48–70, 2017. doi:10.1016/j.ic.2016.03.004.

32 Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999. doi:10.1145/330249.330251.

33 Naoki Kobayashi and Davide Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *ACM Trans. Program. Lang. Syst.*, 32(5):16:1–16:49, 2010. doi:10.1145/1745312.1745313.

34 Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Taking linear logic apart. In Thomas Ehrhard, Maribel Fernández, Valeria de Paiva, and Lorenzo Tortora de Falco, editors, *Proceedings Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity-TLLA@FLoC 2018, Oxford, UK, 7-8 July 2018*, volume 292 of *EPTCS*, pages 90–103, 2018. doi:10.4204/EPTCS.292.5.

35 Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Better late than never: a fully-abstract semantics for classical processes. *Proc. ACM Program. Lang.*, 3(POPL):24:1–24:29, 2019. doi:10.1145/3290337.

36 M.Z. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, 31(7):371–386, 1989. doi:10.1016/0950-5849(89)90159-6.

37 Leslie Lamport. Fairness and hyperfairness. *Distributed Comput.*, 13(4):239–245, 2000. doi:10.1007/PL00008921.

38 Sam Lindley and J. Garrett Morris. Talking bananas: structural recursion for session types. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 434–447. ACM, 2016. doi:10.1145/2951913.2951921.

39 Susan S. Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982. doi:10.1145/357172.357178.

40 Luca Padovani. Fair subtyping for open session types. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, volume 7966 of *Lecture Notes in Computer Science*, pages 373–384. Springer, 2013. doi:10.1007/978-3-642-39212-2_34.

41 Luca Padovani. Deadlock and lock freedom in the linear π-calculus. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 72:1–72:10. ACM, 2014. doi:10.1145/2603088.2603116.

42 Luca Padovani. Fair subtyping for multi-party session types. *Math. Struct. Comput. Sci.*, 26(3):424–464, 2016. doi:10.1017/S096012951400022X.

43 Luca Padovani, Vasco Thudichum Vasconcelos, and Hugo Torres Vieira. Typing liveness in multiparty communicating systems. In eva Kühn and Rosario Pugliese, editors, *Coordination Models and Languages - 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, volume 8459 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2014. doi:10.1007/978-3-662-43376-8_10.

44 Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations for session-based concurrency. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 539–558. Springer, 2012. doi:10.1007/978-3-642-28869-2_27.

45 Jean-Pierre Queille and Joseph Sifakis. Fairness and related properties in transition systems - A temporal logic to deal with fairness. *Acta Informatica*, 19:195–220, 1983. doi:10.1007/BF00265555.

46 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019. doi:10.1145/3290343.

47 Rob van Glabbeek and Peter Höfner. Progress, justness, and fairness. *ACM Comput. Surv.*, 52(4):69:1–69:38, 2019. doi:10.1145/3329125.

48 Rob van Glabbeek, Peter Höfner, and Ross Horne. Assuming just enough fairness to make session types complete for lock-freedom. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470531.

49 Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014. doi:10.1017/S095679681400001X.

50 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In Martín Abadi and Alberto Lluch-Lafuente, editors, *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*, volume 8358 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2013. doi:10.1007/978-3-319-05119-2_3.