Iso-Recursive Multiparty Sessions and their Automated Verification

Marco Giunti[0000−0002−7582−0308] and Nobuko Yoshida[0000−0002−3925−8557]

University of Oxford, UK

Abstract. Most works on session types take an equi-recursive approach and do not distinguish among a recursive type and its unfolding. This becomes more important in recent type systems which do not require global types, also known as *generalised multiparty session types* (GMST). In GMST, in order to establish properties as deadlock-freedom, the environments which type processes are assumed to satisfy extensional properties holding in all infinite sequences. This is a problem because: (1) the mechanisation of GMST and equi-recursion in proof assistants is utterly complex and eventually requires co-induction; and (2) the implementation of GMST in type checkers relies on model checkers for environment verification, and thus the program analysis is not self-contained.
In this paper, we overcome these limitations by providing an *iso-recursive typing system* that computes the *behavioural properties of environments*. The type system relies on a terminating function named *compliance* that computes all final redexes of an environment, and determines when these redexes do not contain mismatches or deadlocks: *compliant environments cannot go wrong*. The function is defined theoretically by introducing the novel notions of deterministic LTS of environments and of environment closure, and can be implemented in mainstream programming languages and compilers. We showcase an implementation in OCaml by using exception handling to tackle the inherent non-determinism of synchronisation of branching and selection types. We assess that the implementation provides the desired properties, namely absence of mismatches and of deadlocks in environments, by resorting to automated deductive verification performed in tools of the OCaml ecosystem relying on Why3.

1 Introduction

Session types [32,51,33] are an effective method to control the behaviour of software components that run in message-passing distributed systems. *Multiparty session types* (MPST) [34,35] enhance session types by providing support for sessions involving multiple participants, thus representing more expressive scenarios. Various theories of MPST have been deployed in programming languages [55] allowing verification of industrial code at compile or run-time [21].

In most works on session types, recursive types follow an *equi-recursive* view [47] and represent infinite trees that are manipulated co-inductively. This representation does not have a direct counterpart in non-lazy programming languages, which typically resort to *iso-recursive* types [1,47] that are manipulated inductively. Moreover, lazy evaluation of predicates on equi-recursive trees might not terminate, and is thus not effective for static program analysis. In practice, MPST are embedded in non-lazy languages by encoding equi-recursive types; for instance, [37] defines infinite sequence of types as polymorphic lenses [20] by using OCaml generalised algebraic data types.

Our proposal to overcome this problem consists in introducing a theory of *iso-recursive multiparty session types* relying on a type system that *computes the deadlock-freedom* of type environments.

Lately, there have been several advances in MPST that can establish deadlock-freedom without using global types, e.g. [38,50,49,5,24,46,30,7]: this bottom-up approach is known as *generalised multiparty session types (GMST)*. However, the price to pay in GMST is that environments must satisfy extensional predicates requiring that a certain property holds for all infinite sequences. This is a non-integrated feature in GMST, which resort to external tools as model checkers to assess these predicates. Moreover, mechanising equi-recursive GMST in proof assistants is quite complex, and eventually relies on co-induction [17]. Specifically, formulations based on GMST are difficult to implement in programming languages because of the interplay among equi-recursive types and the verification of the semantic properties of environments. Another possibility is to proceed top-down by using global types and ensure deadlock freedom without verifying environments, while the analysis' expressiveness is affected by projectability [52].

In this paper, we propose a formal system to compute the deadlock-freedom of type environments *compositionally* at the typing of parallel processes, and we provide an implementation that is automatically verified by using automated deductive tools of the OCaml ecosystem [45,44,9] relying on Why3 [19].

1.1 Equi-recursive vs Iso-recursive Types: SSH/OAuth2 Example

We illustrate our methodology using a recursive variant of the OAuth 2.0 protocol (cf. [49]) which provides support for *ssh* [10]. Let us indicate *send to* and *receive from* participant $\mathtt{p}$ as the *types* $\mathtt{p}!l(S).T$ and $\mathtt{p}?l(S).T$, respectively, where l is a *label* indicating the nature of the communication, S is the *sort* of the payload, and T is the type of the continuation. Selection among (branching on) different output (input) types is done by means of the binary operator $+$. Recursion is provided by the construct $\mu X.T$, which binds the type variable X in T. Termination is represented by type end. *Sorts* describe the types of string, boolean, and unit values. The session types of the service ($\mathtt{s}$), of the client ($\mathtt{c}$), and of the authorisation server ($\mathtt{a}$) are:

$$T_{\mathtt{s}} \overset{\text{def}}{=} \mu X.(\mathtt{c}!\mathsf{login}(\mathsf{unit}).\mathtt{a}?\mathsf{auth}(\mathsf{bool}).X + \mathtt{c}!\mathsf{cancel}(\mathsf{unit}).\mathsf{end})$$

$$T_{\mathtt{c}} \overset{\text{def}}{=} \mu X.(\mathtt{s}?\mathsf{login}(\mathsf{unit}).(\mathtt{a}!\mathsf{pwd}(\mathsf{str}).X + \mathtt{a}!\mathsf{ssh}(\mathsf{unit}).X) + \mathtt{s}?\mathsf{cancel}(\mathsf{unit}).\mathtt{a}!\mathsf{quit}(\mathsf{unit}).\mathsf{end})$$

$$T_{\mathtt{a}} \overset{\text{def}}{=} \mu X.R_{\mathtt{a}}$$

$$R_{\mathtt{a}} \overset{\text{def}}{=} \mathtt{c}?\mathsf{pwd}(\mathsf{str}).\mathtt{s}!\mathsf{auth}(\mathsf{bool}).X + \mathtt{c}?\mathsf{ssh}(\mathsf{unit}).\mathtt{s}!\mathsf{auth}(\mathsf{bool}).X + \mathtt{c}?\mathsf{quit}(\mathsf{unit}).\mathsf{end}$$

The protocol says that the service (s) sends to the client (c) either a request to `login`, or `cancel`; in the first case, c continues by sending the password (`pwd`, carrying a string), or by sending `ssh`, to a, who in turn sends authentication to s (`auth`, with a boolean, telling whether the client is authorised), and the session restarts; in the second case, c sends `quit` to a, and the session ends.

A problem of equi-recursive GMST, e.g. [49], is that types are defined coinductively (cf. [17]). Recursive types can be infinitely folded and unfolded: for instance, we have the following *equi-recursive equations*:

$$T_{\mathsf{a}} = T^{*}_{\mathsf{a}} = T^{**}_{\mathsf{a}} = \cdots$$

$$T^{*}_{\mathsf{a}} \overset{\text{def}}{=} \mathsf{c}?\mathsf{pwd}(\mathsf{str}).\mathsf{s}!\mathsf{auth}(\mathsf{bool}).T_{\mathsf{a}} + \mathsf{c}?\mathsf{ssh}(\mathsf{unit}).\mathsf{s}!\mathsf{auth}(\mathsf{bool}).T_{\mathsf{a}} + \mathsf{c}?\mathsf{quit}(\mathsf{unit}).\mathsf{end}$$

$$T^{**}_{\mathsf{a}} \overset{\text{def}}{=} \mathsf{c}?\mathsf{pwd}(\mathsf{str}).\mathsf{s}!\mathsf{auth}(\mathsf{bool}).T^{*}_{\mathsf{a}} + \mathsf{c}?\mathsf{ssh}(\mathsf{unit}).\mathsf{s}!\mathsf{auth}(\mathsf{bool}).T^{*}_{\mathsf{a}} + \mathsf{c}?\mathsf{quit}(\mathsf{unit}).\mathsf{end}$$

This is particularly relevant when establishing the properties of the typing system, e.g. safety [49, Definition 4.1], which are based on a notion of *transition of session environments*. To illustrate, the idea is to interpret *types as processes*, cf. [41], and consider transitions of *session environments* mapping participants p to types T. The environment $\Gamma_1 \overset{\text{def}}{=} \mathsf{s}\colon \mathsf{c}!\mathsf{login}(\mathsf{unit}).\mathsf{a}?\mathsf{auth}(\mathsf{bool}).T_{\mathsf{s}} + \mathsf{c}!\mathsf{cancel}(\mathsf{unit}).\mathsf{end}$ can fire an output action $\mathsf{c}!\mathsf{login}(\mathsf{unit})$ and reach $\mathsf{s}\colon \mathsf{a}?\mathsf{auth}(\mathsf{bool}).T_{\mathsf{s}}$, or can fire an output action $\mathsf{c}!\mathsf{cancel}(\mathsf{unit})$ and reach $\mathsf{s}\colon \mathsf{end}$. The environment $\Gamma_2 \overset{\text{def}}{=} \mathsf{c}\colon \mathsf{s}?\mathsf{login}(\mathsf{unit}).T'_{\mathsf{c}} + \mathsf{s}?\mathsf{cancel}(\mathsf{unit}).\mathsf{a}!\mathsf{quit}(\mathsf{unit}).\mathsf{end}$ can fire an input action $\mathsf{s}?\mathsf{login}(\mathsf{unit})$ and reach $\mathsf{c}\colon T'_{\mathsf{c}}$, where $T'_{\mathsf{c}} \overset{\text{def}}{=} \mathsf{a}!\mathsf{pwd}(\mathsf{str}).T_{\mathsf{c}} + \mathsf{a}!\mathsf{ssh}(\mathsf{unit}).T_{\mathsf{c}}$, or can fire an input action $\mathsf{s}?\mathsf{cancel}(\mathsf{unit})$ and reach $\mathsf{c}\colon \mathsf{a}!\mathsf{quit}(\mathsf{unit}).\mathsf{end}$. The environment Γ_1, Γ_2 can fire two synchronisation actions: (1) $\mathsf{login}@\mathsf{s}\bowtie\mathsf{c}$, which indicates a synchronisation on the label login by the input participant s and the output participant c, and reach the environment $\mathsf{s}\colon \mathsf{a}?\mathsf{auth}(\mathsf{bool}).T_{\mathsf{s}}, \mathsf{c}\colon T'_{\mathsf{c}}$; and (2) $\mathsf{cancel}@\mathsf{s}\bowtie\mathsf{c}$, and reach the environment $\mathsf{s}\colon \mathsf{end}, \mathsf{c}\colon \mathsf{a}!\mathsf{quit}(\mathsf{unit}).\mathsf{end}$.

In particular, the rule for recursive types in [49, Definition 2.8] states that a recursive type $\mu X.T$ inherits the transitions from its unfolding, that is the type $T\{\mu X.T/X\}$. For instance, the rule can be instantiated with type T_{a} as

$$[\Gamma\text{-}\mu]\,\frac{\Gamma, \mathsf{a}\colon T^{*}_{\mathsf{a}} \xrightarrow{\alpha} \Gamma'}{\Gamma, \mathsf{a}\colon T_{\mathsf{a}} \xrightarrow{\alpha} \Gamma'}$$

and allows for inferring the following transitions:

$$\Gamma, \mathsf{a}\colon T_{\mathsf{a}} \xrightarrow{\mathtt{c?ssh(unit)}} \Gamma, \mathsf{a}\colon \mathsf{s}!\mathsf{auth}(\mathsf{bool}).T_{\mathsf{a}}$$
$$\Gamma, \mathsf{a}\colon T_{\mathsf{a}} \xrightarrow{\mathtt{c?ssh(unit)}} \Gamma, \mathsf{a}\colon \mathsf{s}!\mathsf{auth}(\mathsf{bool}).T^{*}_{\mathsf{a}} \quad \cdots$$

We note that this elegant approach is appropriate for the theory, but less suited for mechanising GMST in theorem provers, and for automated verification.

More specifically, the approach introduced in [49] and followed in many subsequent papers on GMST, e.g. [38,49,5,24,46,30,7], requires to type check sessions with environments having certain *extensional properties*. Crucially, such properties must be established *before typing* by analysing all possible infinite transitions of session environments. To illustrate, the paper [49] provides a companion artefact by using mCRL2 [29] featuring μ-calculus formulae that represent the safety and deadlock freedom properties of environments [49, Figure 5], which are defined by least and greatest fixed points.

Our solution. In this paper, we *compute the semantic properties* of the session environment in the rule for *type checking the session composition*, hence achieving *decidable type checking*. This is possible because our types are iso-recursive and have a finite structure.

In our setting, the types T_{a}, T^*_{a}, and T^{**}_{a} are all different, but isomorphic. A recursive type $\mu X.T$ can only be used to type-check a recursive process, or a type variable. To type check an input or output process, we need to unfold $\mu X.T$ by applying the substitution $\mu X.T/X$ to type T, denoted as $T\{\mu X.T/X\}$. That is, we have the following *iso-recursive equations*:

$$T^*_{\mathsf{a}} = R_{\mathsf{a}}\{T_{\mathsf{a}}/X\} \qquad T^{**}_{\mathsf{a}} = R_{\mathsf{a}}\{T^*_{\mathsf{a}}/X\} = R_{\mathsf{a}}\{(R_{\mathsf{a}}\{T_{\mathsf{a}}/X\})/X\} \cdots$$

For instance, consider the authorisation process Q_{a} below, whose syntax mirrors the one of its type T^*_{a}, but that the payload of input and output are (bound) variables and expressions, respectively, and that there is a process variable χ:

$$Q_{\mathsf{a}} \stackrel{\text{def}}{=} \mathsf{c}?\mathsf{pwd}(x).\mathsf{s}!\mathsf{auth}\langle\mathsf{false}\rangle.\chi + \mathsf{c}?\mathsf{ssh}(x).\mathsf{s}!\mathsf{auth}\langle\mathsf{true}\rangle.\chi + \mathsf{c}?\mathsf{quit}(x).\mathbf{0}$$

In order to type check the recursive process $\mu\chi.Q_{\mathsf{a}}$ we use the typing judgement:

$$\text{T-REC}\,\frac{\chi\colon T_{\mathsf{a}} \vdash Q_{\mathsf{a}} : T^*_{\mathsf{a}}}{\emptyset \vdash \mu\chi.Q_{\mathsf{a}} : T_{\mathsf{a}}}$$

Now consider the process obtained by substituting χ in Q_{a} with $\mu\chi.Q_{\mathsf{a}}$, that is process $P^*_{\mathsf{a}} \stackrel{\text{def}}{=} Q_{\mathsf{a}}\{\mu\chi.Q_{\mathsf{a}}/\chi\}$, and the parallel execution of $\mathsf{a} \lhd P^*_{\mathsf{a}}$ with $\mathsf{s} \lhd P_{\mathsf{s}}$ and $\mathsf{c} \lhd P_{\mathsf{c}}$, where $P_{\mathsf{s}}, P_{\mathsf{c}}$ are recursive processes implementing the service s typed by T_{s}, and the client c typed by T_{c}, respectively. This session should be accepted by the type system, since at runtime it behaves correctly, independently of the fact that the authorisation service P^*_{a} has been "unrolled" once.

That is, we want to infer the following judgement by using the rule for session composition of the typing system for sessions, denoted $\Vdash$:

$$\text{T-SES}\,\frac{\emptyset \vdash P_{\mathsf{s}} : T_{\mathsf{s}} \quad \emptyset \vdash P_{\mathsf{c}} : T_{\mathsf{c}} \quad \emptyset \vdash P^*_{\mathsf{a}} : T^*_{\mathsf{a}} \quad \Delta = \mathsf{s} : T_{\mathsf{s}}, \mathsf{c} : T_{\mathsf{c}}, \mathsf{a} : T^*_{\mathsf{a}} \quad \mathsf{comp}(\Delta)}{\emptyset \Vdash \mathsf{s} \lhd P_{\mathsf{s}} \parallel \mathsf{c} \lhd P_{\mathsf{c}} \parallel \mathsf{a} \lhd P^*_{\mathsf{a}} \rhd \Delta}$$

The predicate $\mathsf{comp}(\Delta)$ establishes *compliance* by using the *computable function* comp. The goal is to calculate all possible *final environments* that are reachable from Δ, and verify that they are not errors. Intuitively, an environment is final when is stuck, or when it has already been encountered, reaching a fixed point.

Since we are interested in mechanising compliance, the calculation should be achieved by relying on the novel notion of *deterministic transition*, denoted $\longrightarrow_d$, such that $\Delta \xrightarrow{\alpha_1}_d \Delta_1$ and $\Delta \xrightarrow{\alpha_2}_d \Delta_2$ imply $\alpha_1 = \alpha_2$ and $\Delta_1 = \Delta_2$. The key point is that a deterministic transition system can be encoded as a computable function that can be deployed in type checkers and compilers. Moreover, the properties of the function can be verified with automated deductive verification tools as Why3 [19]. In particular, we propose the idea of *closure of an environment* Δ: the function receives Δ in input and returns in output a finite set of final environments reachable from Δ by multiple applications of $\longrightarrow_d$.

The compliance function decides when in all final environments reached by transitions starting from Δ, there is not a *communication mismatch* or a *deadlock*. A communication mismatch arises when a participant $\mathtt{p}$ has a single I/O type receiving from/sending to participant $\mathtt{q}$, and $\mathtt{q}$ has a single I/O type receiving from/sending to participant $\mathtt{p}$, and one of the following cases arise: (i) both $\mathtt{p}$ and $\mathtt{q}$ are sending or receiving; (ii) the intersection among the labels used by $\mathtt{p}$ and $\mathtt{q}$ is empty; (iii) $\mathtt{p}$ and $\mathtt{q}$ agree on a label but disagree on the label's sort. A deadlock arises when the environment Δ cannot fire any transition and there is at least a participant $\mathtt{p}$ s.t. $\Delta(\mathtt{p}) \neq \mathsf{end}$.

To see an example of environment rejected by the compliance function comp, consider Δ'' below. An authorisation server typed by $T''_{\mathtt{a}}$ only allows two subsequent attempts for `ssh` authentication: after that, it ends. Conversely, a client typed by $T_{\mathtt{c}}$ performs an infinite number of requests of `ssh` authentication: for this very reason, a system typed by Δ'' can deadlock and must be rejected.

$$\begin{aligned} T'_{\mathtt{a}} &\stackrel{\text{def}}{=} \mu X.(\mathtt{c}?\mathsf{pwd}(\mathsf{str}).\mathtt{s}!\mathsf{auth}(\mathsf{bool}).X + \mathtt{c}?\mathsf{ssh}(\mathsf{unit}).\mathtt{s}!\mathsf{auth}(\mathsf{bool}).\mathsf{end} + \\ &\qquad \mathtt{c}?\mathsf{quit}(\mathsf{unit}).\mathsf{end}) \\ T''_{\mathtt{a}} &\stackrel{\text{def}}{=} \mathtt{c}?\mathsf{pwd}(\mathsf{str}).\mathtt{s}!\mathsf{auth}(\mathsf{bool}).T_{\mathtt{a}} + \mathtt{c}?\mathsf{ssh}(\mathsf{unit}).\mathtt{s}!\mathsf{auth}(\mathsf{bool}).T'_{\mathtt{a}} + \mathtt{c}?\mathsf{quit}(\mathsf{unit}).\mathsf{end} \\ \Delta'' &\stackrel{\text{def}}{=} \mathtt{s}\colon T_{\mathtt{s}}, \mathtt{c}\colon T_{\mathtt{c}}, \mathtt{a}\colon T''_{\mathtt{a}} \end{aligned}$$

The closure of Δ'' does return a set of environments containing the deadlocked environment $\Delta_{\mathtt{lock}}$, which depicts the scenario discussed above. Since $\Delta_{\mathtt{lock}} \in \mathsf{closure}(\Delta'')$ and $\Delta_{\mathtt{lock}}$ is a deadlock, we have $\neg\,\mathsf{comp}(\Delta'')$:

$$\Delta_{\mathtt{lock}} \stackrel{\text{def}}{=} \mathtt{s}\colon \mathtt{a}?\mathsf{auth}(\mathsf{bool}).T_{\mathtt{s}}, \mathtt{c}\colon \mathtt{a}!\mathsf{pwd}(\mathsf{str}).T_{\mathtt{c}} + \mathtt{a}!\mathsf{ssh}(\mathsf{unit}).T_{\mathtt{c}}, \mathtt{a}\colon \mathsf{end}$$

Outline. **§ 2** introduces the syntax and semantics of multiparty sessions. **§ 3** presents the non-deterministic labelled transition semantics of session environments (cf. **§ 3.1**), and its deterministic counterpart (cf. **§ 3.2**): the former is used to define deadlocks and to prove subject reduction; the latter is used in **§ 3.3** to define closure and in turn to mechanise compliance. **§ 4** introduces the typing system. We first analyse the typing rules for processes. Second, we analyse the rule for typing sessions, which relies on a computable function calculating compliance that is defined in **§ 4.1**. Last, in **§ 4.2** we provide the proof of subject reduction and we state a progress result. **§ 5** is devoted to the automated deductive verification of compliance. We start in **§ 5.1** by outlining few details of the implementation of compliance and of closure of deterministic transitions in OCaml. **§ 5.2** verifies the behavioural specification of the implementation in automated deductive verification tools of the OCaml ecosystem relying on Why3. **§ 6** concludes by presenting related work and next directions. The full proofs and omitted definitions can be found in [27] and the accompanying artefact can be found at `https://doi.org/10.5281/zenodo.14621028`.

2 Multiparty Sessions

The syntax of types and processes is in Definition 1. We consider *iso-recursive* types of the form $\mu X.T$ where $\mu X.T$ and its unfolding are not equal, but isomorphic. We stress that types have a *finite representation* rather than abstract infinite trees (cf. equi-recursive types).

Definition 1 (Syntax of types and processes).

$$\begin{array}{rll}
S := & \mathsf{nat} \mid \mathsf{int} \mid \mathsf{str} \mid \mathsf{bool} \mid \mathsf{unit} & \text{Sorts}\\
T := & \mathtt{r}!l(S).T \mid \mathtt{r}?l(S).T \mid T + T \mid \mathsf{end} \mid \mu X.T \mid X & \text{Types}\\
P := & \mathtt{r}!l\langle e\rangle.P \mid \mathtt{r}?l(x).P \mid P + P \mid \mu\chi.P \mid \chi \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid \mathbf{0} & \text{Processes}\\
\mathcal{M} := & \mathtt{p} \triangleleft P \mid \|_{i\in I}\ \mathtt{p}_i \triangleleft P_i & \text{Sessions}
\end{array}$$

We require all terms to be contractive, i.e. $\mu X_1.\mu X_2.\ldots\mu X_n.X_1$ is not allowed as a sub-term for any $n \geq 1$ [47, p. 300], which can be alternatively stated as type variables occur guarded (by input or output prefixes) [14].[1]

We use $\mathtt{p}, \mathtt{q}, \mathtt{r}$ to range over *participants*, l to range over *labels*, and i, j to range over indexes (natural numbers). X, Y range over *type variables*, e, e' range over *expressions*, v, w range over *values*, x, y range over *variables*, and χ range over *process variables*. *Sessions* $\mathcal{M}$ belong to the set $\mathbb{M}$. A single session or *thread* is a process P indexed by a participant, denoted $\mathtt{p} \triangleright P$. A *multiparty session* is a composition of all threads, denoted $\|_{i\in I}\ \mathtt{p}_i \triangleleft P_i$ or $\mathtt{p}_1 \triangleleft P_1 \parallel \cdots \parallel \mathtt{p}_n \triangleleft P_n$.

The constructor μ is a *binder* in types and processes, respectively: we let X be bound in $\mu X.T$ and *free* in T; similarly, χ is bound in $\mu\chi.P$ and free in P. The remaining binder for processes is input: variable x is bound in $\mathtt{r}?l(x).P$ and free in P. *Closed* terms are those without free variables.

We assume the *substitution* of free occurrences of a type variable X in a type T_1 with a closed type T_2, written $T_1\{T_2/X\}$. We assume the substitution of free occurrences of a process variable χ in process P_1 with a closed process P_2, written $P_1\{P_2/\chi\}$, and the substitution of free occurrences of variable x in process P with a value v, written $P\{v/x\}$. A type R is *μ-guarded* (*guarded*, for short) if it is a sub-term of T in the definition $\mu X.T$.

The symbol $=$ is reserved for Leibniz equality.

Definition 2 (Session notation).

$$\begin{array}{lll}
\oplus_{i\in I}\mathtt{r}!l_i(S_i).T_i & \overset{def}{=} \mathtt{r}!l_1(S_1).T_1 + \cdots + \mathtt{r}!l_n(S_n).T_n & I = (1,\ldots,n), n \geq 1\\
\&_{i\in I}\mathtt{r}?l_i(S_i).T_i & \overset{def}{=} \mathtt{r}?l_1(S_1).T_1 + \cdots + \mathtt{r}?l_n(S_n).T_n & I = (1,\ldots,n), n \geq 1
\end{array}$$

The next step towards the definition of the typing system is to identify *well-formed* types that correctly abstract multiparty sessions. The definition is in the technical report [27]. We collect the labels of types in multi-sets, and the polarities and the participants of types in sets. Intuitively, a sum type $T_1 + T_2$ is *well-behaved* when it has not duplicated labels, T_1 and T_2 have the same

[1] Formally, contractiveness is mechanised in Coq [11] by relying on the reflexive-transitive closure of the transition system of types introduced in § 3.

$$\text{R-INP}\ \dfrac{}{\mathtt{p} \triangleleft \mathtt{q}?l(x).P \xrightarrow{\mathtt{q}?l(v)} \mathtt{p} \triangleleft P\{v/x\}} \qquad \text{R-OUT}\ \dfrac{e \downarrow v}{\mathtt{p} \triangleleft \mathtt{q}!l\langle e\rangle.P \xrightarrow{\mathtt{q}!l\langle v\rangle} \mathtt{p} \triangleleft P}$$

$$\text{R-SUM-L}\ \dfrac{\mathtt{r} \triangleleft P \xrightarrow{\alpha} \mathtt{r} \triangleleft P'}{\mathtt{r} \triangleleft P + Q \xrightarrow{\alpha} \mathtt{r} \triangleleft P'}$$

$$\text{R-COM}\ \dfrac{\mathtt{p} \triangleleft P \xrightarrow{\mathtt{q}?l(v)} \mathtt{p} \triangleleft P' \qquad \mathtt{q} \triangleleft Q \xrightarrow{\mathtt{p}!l\langle v\rangle} \mathtt{q} \triangleleft Q'}{\mathtt{p} \triangleleft P \parallel \mathtt{q} \triangleleft Q \parallel_{i\in I} \mathtt{r}_i \triangleleft R_i \xrightarrow{l@\mathtt{p}\bowtie\mathtt{q}} \mathtt{p} \triangleleft P' \parallel \mathtt{q} \triangleleft Q' \parallel_{i\in I} \mathtt{r}_i \triangleleft R_i}$$

$$\text{R-REC}\ \dfrac{}{\mathtt{r} \triangleleft \mu\chi.P \parallel_{i\in I} \mathtt{r}_i \triangleleft R_i \xrightarrow{\tau} \mathtt{r} \triangleleft P\{\mu\chi.P/\chi\} \parallel_{i\in I} \mathtt{r}_i \triangleleft R_i}$$

$$\text{R-IFT}\ \dfrac{e \downarrow \mathsf{true}}{\mathtt{r} \triangleleft \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \parallel_{i\in I} \mathtt{r}_i \triangleleft R_i \xrightarrow{\tau} \mathtt{r} \triangleleft P \parallel_{i\in I} \mathtt{r}_i \triangleleft R_i}$$

$$\text{R-STR}\ \dfrac{\mathcal{M}_1' \equiv \mathcal{M}_1 \qquad \mathcal{M}_1 \xrightarrow{\alpha} \mathcal{M}_2 \qquad \mathcal{M}_2 \equiv \mathcal{M}_2'}{\mathcal{M}_1' \xrightarrow{\alpha} \mathcal{M}_2'}$$

Fig. 1. Labelled transition rules for multiparty sessions (we omit R-IFF)

unique polarity, and the same unique participant. These assumptions eliminate ill-types of the form e.g. $\mathtt{p}!l(S_1).T_1 + \mathtt{p}!l(S_2).T_2$ or of the form e.g. $\mathtt{p}_1?l_1(S_1).T_1 + \mathtt{p}_2?l_2(S_2).T_2$ with $\mathtt{p}_1 \neq \mathtt{p}_2$, as well as mixed choice types, e.g. $\mathtt{p}!l_1(S_1).T_1 + \mathtt{p}?l_2(S_2).T_2$. A type T is well-formed, denoted $\mathrm{WF}(T)$, when it is well-behaved, contractive, and closed.

Operational semantics of multiparty sessions. We assume an *evaluation* function $\downarrow$ transforming expressions e into boolean, integer and unit values v, written $e \downarrow v$. The operational semantics of multiparty sessions are defined modulo a *structural congruence* relation over sessions $\mathcal{M}$, denoted $\equiv \subseteq \mathbb{M} \times \mathbb{M}$. We let $\equiv$ be the least reflexive relation that satisfies the axiom

$$\parallel_{i\in I} \mathtt{p}_i \triangleleft P_i \equiv \parallel_{j\in J} \mathtt{p}_j \triangleleft P_j \qquad (\mathsf{permutation}(I, J))$$

The *labelled transition rules* are defined in Figure 1; we just present the left rules. A *computation* is a sequence of α-transitions, $\alpha \in \{\tau, l@\mathtt{p}\bowtie\mathtt{q}\}$, or *reductions* $\mathcal{M}_1 \xrightarrow{\alpha} \mathcal{M}_2 \xrightarrow{\alpha} \cdots$. We are mainly interested in analysing computations of well-typed sessions (cf. § 4).

Rule R-INP says that a participant $\mathtt{p}$ waiting for a value from $\mathtt{q}$ on the label l can do a transition labelled by $\mathtt{q}?l(v)$ and instantiate the formal parameter x with the value v in the continuation P, noted as $P\{v/x\}$. Rule R-OUT allows a participant $\mathtt{p}$ sending to $\mathtt{q}$ on label l an expression e that can be evaluated as v to do a transition labelled by $\mathtt{q}!l\langle v\rangle$ and continue as P. Non-deterministic reductions are allowed by means of rule R-SUM-L, which says that a participant $\mathtt{r}$

non-deterministically choosing among process P and Q, denoted $P+Q$, can do a transition labelled by α and reach $\mathtt{r} \lhd P'$ whenever $\mathtt{r} \lhd P$ can fire the same transition and reach the same redex.

Communication among two participants $\mathtt{p}$ and $\mathtt{q}$ is performed by means of rule R-COM. Whenever $\mathtt{p} \lhd P$ can do a transition labelled by the input action $\mathtt{q}?l(v)$ and reach the redex $\mathtt{p} \lhd P'$, and $\mathtt{q} \lhd Q$ can do a transition labelled by the output action $\mathtt{p}!l\langle v\rangle$ and reach the redex $\mathtt{q} \lhd Q'$, we can infer a transition labelled with $l@\mathtt{p}\bowtie\mathtt{q}$ from the composition of $\mathtt{p} \lhd P$ and $\mathtt{q} \lhd Q$ and a session $\|_{i\in I}\ \mathtt{r}_i \lhd R_i$ to the composition of $\mathtt{p} \lhd P'$ and $\mathtt{q} \lhd Q'$ and $\|_{i\in I}\ \mathtt{r}_i \lhd R_i$. Rule R-REC allows a participant $\mathtt{r}$ recursively defined as $\mu\chi.P$ and running in parallel with a session $\|_{i\in I}\ \mathtt{r}_i \lhd R_i$, to do an internal transition τ and unfold the body P while instantiating the occurrences of χ in P with $\mu\chi.P$, thus reaching the redex $\mathtt{r} \lhd P\{\mu\chi.P/\chi\}\ \|_{i\in I}\ \mathtt{r}_i \lhd R_i$. Rule R-IFT (R-IFF) says that a participant $\mathtt{r}$ with the body $\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$ and running in parallel with a session $\|_{i\in I}\ \mathtt{r}_i \lhd R_i$, can do a τ-transition and reach the redex $\mathtt{r} \lhd P\ \|_{i\in I}\ \mathtt{r}_i \lhd R_i$ ($\mathtt{r} \lhd Q\ \|_{i\in I}\ \mathtt{r}_i \lhd R_i$) whenever the expression e evaluates to true (false). Rule R-STR rearranges processes with structural congruence.

Example 1. Consider the authorisation protocol in § 1.1 and

$$\begin{aligned}
Q_{\mathtt{a}} &\stackrel{\text{def}}{=} \mathtt{c}?\mathsf{pwd}(x).\mathtt{s}!\mathsf{auth}\langle\mathsf{false}\rangle.\chi + \mathtt{c}?\mathsf{ssh}(x).\mathtt{s}!\mathsf{auth}\langle\mathsf{true}\rangle.\chi + \mathtt{c}?\mathsf{quit}(x).\mathbf{0}\\
P_{\mathtt{s}} &\stackrel{\text{def}}{=} \mu\chi.(\mathtt{c}!\mathsf{login}\langle\rangle.\mathtt{a}?\mathsf{auth}(x).\chi + \mathtt{c}!\mathsf{cancel}\langle\rangle.\mathbf{0})\\
P_{\mathtt{c}} &\stackrel{\text{def}}{=} \mu\chi.(\mathtt{s}?\mathsf{login}(x).(\mathtt{a}!\mathsf{pwd}\langle\text{“fido”}\rangle.\chi + \mathtt{a}!\mathsf{ssh}\langle\rangle.\chi)\ + \mathtt{s}?\mathsf{cancel}(x).\mathtt{a}!\mathsf{quit}\langle\rangle.\mathbf{0})\\
\mathcal{M} &\stackrel{\text{def}}{=} \mathtt{s} \lhd P_{\mathtt{s}} \parallel \mathtt{c} \lhd P_{\mathtt{c}} \parallel \mathtt{a} \lhd P^*_{\mathtt{a}}
\end{aligned}$$

where process $P^*_{\mathtt{a}} \stackrel{\text{def}}{=} Q_{\mathtt{a}}\{\mu\chi.Q_{\mathtt{a}}/\chi\}$ implements the (unfolding of the) authorisation server $\mathtt{a}$, and processes $P_{\mathtt{s}}$ and $P_{\mathtt{c}}$ implement the service $\mathtt{s}$ and the client $\mathtt{c}$, respectively. We analyse transitions of the session introduced in § 1.1 and composing the service $\mathtt{s}$, the client $\mathtt{c}$, and the server $\mathtt{a}$, here referred as $\mathcal{M}$.

We want to analyse a communication of the server $\mathtt{s}$ with the client $\mathtt{c}$ depicting a `login` transaction. A first application of rule R-REC unfolds the service $\mathtt{s}$:

$$\mathcal{M} \xrightarrow{\tau} \mathtt{s} \lhd P^*_{\mathtt{s}} \parallel \mathtt{c} \lhd P_{\mathtt{c}} \parallel \mathtt{a} \lhd P^*_{\mathtt{a}} \stackrel{\text{def}}{=} \mathcal{M}_1$$

where $P^*_{\mathtt{s}} \stackrel{\text{def}}{=} \mathtt{c}!\mathsf{login}\langle\rangle.\mathtt{a}?\mathsf{auth}(x).P_{\mathtt{s}} + \mathtt{c}!\mathsf{cancel}\langle\rangle.\mathbf{0}$.

The next step consists in unfolding the client $\mathtt{c}$. Since the client thread does not occur in the left, we need to first apply R-REC and then apply structural congruence in rule R-STR:

$$\text{R-STR}\ \dfrac{\text{R-REC}\ \dfrac{}{\mathtt{c} \lhd P_{\mathtt{c}} \parallel \mathtt{s} \lhd P^*_{\mathtt{s}} \parallel \mathtt{a} \lhd P^*_{\mathtt{a}} \xrightarrow{\tau} \mathtt{c} \lhd P^*_{\mathtt{c}} \parallel \mathtt{s} \lhd P^*_{\mathtt{s}} \parallel \mathtt{a} \lhd P^*_{\mathtt{a}}}}{\mathcal{M}_1 \xrightarrow{\tau} \mathtt{s} \lhd P^*_{\mathtt{s}} \parallel \mathtt{c} \lhd P^*_{\mathtt{c}} \parallel \mathtt{a} \lhd P^*_{\mathtt{a}} \stackrel{\text{def}}{=} \mathcal{M}_2}$$

where $P^*_{\mathtt{c}} \stackrel{\text{def}}{=} \mathtt{s}?\mathsf{login}(x).(\mathtt{a}!\mathsf{pwd}\langle\text{“fido”}\rangle.P_{\mathtt{c}} + \mathtt{a}!\mathsf{ssh}\langle\rangle.P_{\mathtt{c}})\ + \mathtt{s}?\mathsf{cancel}(x).\mathtt{a}!\mathsf{quit}\langle\rangle.\mathbf{0}$. Now we apply rule R-COM to infer a communication among the service $\mathtt{s}$ and the client $\mathtt{c}$ on the label `login`, followed by R-STR:

$$\textsc{R-Str}\;\dfrac{\textsc{R-Com}\;\dfrac{(A)\qquad(B)}{\mathsf{c} \lhd P^*_{\mathsf{c}} \parallel \mathsf{s} \lhd P^*_{\mathsf{s}} \parallel \mathsf{a} \lhd P^*_{\mathsf{a}} \xrightarrow{\texttt{login}@\mathsf{c}\bowtie\mathsf{s}} \mathsf{c} \lhd P'_{\mathsf{c}} \parallel \mathsf{s} \lhd P'_{\mathsf{s}} \parallel \mathsf{a} \lhd P^*_{\mathsf{a}}}}{\mathcal{M}_2 \xrightarrow{\texttt{login}@\mathsf{c}\bowtie\mathsf{s}} \mathcal{M}_3}$$

where $\mathcal{M}_3 \overset{\text{def}}{=} \mathsf{s} \lhd \mathsf{a}?\mathsf{auth}(x).P_{\mathsf{s}} \parallel \mathsf{c} \lhd P'_{\mathsf{c}} \parallel \mathsf{a} \lhd P^*_{\mathsf{a}}$, $P'_{\mathsf{s}} \overset{\text{def}}{=} \mathsf{a}?\mathsf{auth}(x).P_{\mathsf{s}}$, $P'_{\mathsf{c}} \overset{\text{def}}{=} \mathsf{a}!\mathsf{pwd}\langle\text{“fido”}\rangle.P_{\mathsf{c}} + \mathsf{a}!\mathsf{ssh}\langle\rangle.P_{\mathsf{c}}$, and

$$(A)\;\textsc{R-Sum-L}\;\dfrac{\textsc{R-Inp}\;\dfrac{}{\mathsf{s}?\mathsf{login}(x).P'_{\mathsf{c}} \xrightarrow{\mathsf{s}?\texttt{login}()} P'_{\mathsf{c}}}}{\mathsf{c} \lhd P^*_{\mathsf{c}} \xrightarrow{\mathsf{s}?\texttt{login}()} \mathsf{c} \lhd P'_{\mathsf{c}}}$$

$$(B)\;\textsc{R-Sum-L}\;\dfrac{\textsc{R-Out}\;\dfrac{}{\mathsf{s} \lhd \mathsf{c}!\mathsf{login}\langle\rangle.P'_{\mathsf{s}} \xrightarrow{\mathsf{c}!\texttt{login}\langle\rangle} \mathsf{s} \lhd P'_{\mathsf{s}}}}{\mathsf{s} \lhd P^*_{\mathsf{s}} \xrightarrow{\mathsf{c}!\texttt{login}\langle\rangle} \mathsf{s} \lhd P'_{\mathsf{s}}}$$

As you can see, in session $\mathcal{M}_3$ the client c is ready to communicate the *password*, or to send a `ssh` request, to the authorisation server a. $\square$

3 Session Environment Reduction, Algorithmically

A central notion of multiparty session types is the interaction among parties. We model this abstraction by depicting the behaviour of *session environments* Δ assigning types T to participants p.

Our aim is to define a *function* that decides at *compile-time* when it is safe to type-check a group of participants running in parallel and willing to communicate with each other. This is reminiscent of the notion of type duality in binary session types (e.g. [22,26]), but encompasses multiple participants. We will use the function in the typing system introduced in § 4.

Definition 3 (Labelled transition system). *A labelled transition system (LTS) is a tuple $(\tilde{A}, \mathcal{S}_1, \mathcal{A}, \mathcal{S}_2, \rightarrow)$, noted as $\tilde{A} \triangleright \sigma_1 \xrightarrow{\alpha} \sigma_2$, whenever $\sigma_1 \in \mathcal{S}_1$ and $\sigma_2 \in \mathcal{S}_2$ and $\alpha \in \mathcal{A}$, where $\tilde{A}$ is a (possibly empty) tuple of parameters, $\mathcal{S}_i$ are set of states, $i = 1, 2$, $\mathcal{A}$ is a set of actions, and $\rightarrow$ is a transition relation s.t. $\rightarrow \subseteq \tilde{A} \times \mathcal{S}_1 \times \mathcal{A} \times \mathcal{S}_2$. A transition relation $\rightarrow$ is a partial function whenever $\tilde{A} \triangleright \sigma_1 \xrightarrow{\alpha'} \sigma'_2$ and $\tilde{A} \triangleright \sigma_1 \xrightarrow{\alpha''} \sigma''_2$ imply $\alpha' = \alpha''$ and $\sigma'_2 = \sigma''_2$. A LTS is deterministic whenever its transition relation is a partial function.*

3.1 Non-deterministic Transition System

We first define a non-deterministic LTS of session environments, and then in § 3.2 we outline its transformation to a deterministic LTS. Non-deterministic transitions are used in the notion of deadlock (cf. Definition 10), and in the proof of

Transition rules for types: $\boxed{T \xrightarrow{\alpha} T}$

$$\text{E-Out}\ \frac{\emptyset \vdash v \colon S}{\mathtt{r}!l(S).T \xrightarrow{\mathtt{r}!l\langle v\rangle} T} \qquad \text{E-In}\ \frac{\emptyset \vdash v \colon S}{\mathtt{r}?l(S).T \xrightarrow{\mathtt{r}?l(v)} T} \qquad \text{E-Sel-L}\ \frac{T_1 \xrightarrow{\mathtt{r}!l\langle v\rangle} T'}{T_1 + T_2 \xrightarrow{\mathtt{r}!l\langle v\rangle} T'}$$

$$\text{E-Bra-L}\ \frac{T_1 \xrightarrow{\mathtt{r}?l(v)} T'}{T_1 + T_2 \xrightarrow{\mathtt{r}?l(v)} T'} \qquad \text{E-Rec}\ \frac{}{\mu X.T \xrightarrow{\tau} T\{\mu X.T/X\}}$$

Transition rules for session environments: $\boxed{\Delta \xrightarrow{\alpha} \Delta}$

$$\text{Se-Rec}\ \frac{T \xrightarrow{\tau} T'}{\Delta, \mathtt{p} : T \xrightarrow{\tau} \Delta, \mathtt{p} : T'} \qquad \text{Se-Com}\ \frac{T_{\mathtt{p}} \xrightarrow{\mathtt{q}?l(v)} T'_{\mathtt{p}} \qquad T_{\mathtt{q}} \xrightarrow{\mathtt{p}!l\langle v\rangle} T'_{\mathtt{q}}}{\Delta, \mathtt{p} : T_{\mathtt{p}}, \mathtt{q} : T_{\mathtt{q}} \xrightarrow{l@\mathtt{p}\bowtie\mathtt{q}} \Delta, \mathtt{p} : T'_{\mathtt{p}}, \mathtt{q} : T'_{\mathtt{q}}}$$

Transition rule for configurations: $\boxed{D \diamond \Delta \xrightarrow{\alpha} D \diamond \Delta}$

$$\text{Se-Top}\ \frac{\Delta \in D \qquad \Delta \xrightarrow{\alpha} \Delta'}{D \diamond \Delta \xrightarrow{\alpha} D\backslash_{\Delta} \diamond \Delta'}$$

Fig. 2. Labelled transition system of session environments

subject reduction (cf. § 4.2). In the non-deterministic setting, the parameters $\widetilde{A}$ are empty and $\mathcal{S}_1 = \mathcal{S}_2$.

We start by defining a non-deterministic LTS of types. Since we will also use the transition system to match the actions of processes, it is practical to use the same labels of the LTS of Figure 1. The left rules for types are in Figure 2. The rules are designed for well-formed types (cf. § 2), as we discuss below (cf. rules E-Sel-L, fitsE-Bra-L). Rule E-Out says that a type doing an output to the participant $\mathtt{r}$ on label l with payload S and continuing as T can fire the action $\mathtt{r}!l\langle v\rangle$ and reach the redex T whenever v is a value of sort S. Dually, rule E-In allows an input type from $\mathtt{r}$ on label l with payload S and continuing as T to do an action $\mathtt{r}?l(v)$ and reach the redex T, if v has sort S. Rule E-Sel-L allows a sum type $T_1 + T_2$ to do an output action $\mathtt{r}!l\langle v\rangle$ and reach the redex T' whenever T_1 can fire this action and reach T'. Dually, rule E-Bra-L allows a sum type $T_1 + T_2$ to do an input action $\mathtt{r}?l(v)$ and reach the redex T' if T_1 can fire this action and reach T'. Note that input and output are the only actions that a sum type can fire. This is because types as e.g. $T_1 + \mu X.T$ or $T_1 + (\mu X.T + T_2)$ are not well-formed.

The non-deterministic transition rules for session environments follow in Figure 2, and are the counterpart of the non-deterministic rules of the form $\Gamma \xrightarrow{\alpha} \Gamma$ used in GMST (cf. [49]) to analyse the *safety* and *deadlock freedom* of multiparty protocols. We consider a top-level rule of the form $D \diamond \Delta \xrightarrow{\alpha} D \diamond \Delta$, where we refer to $D \diamond \Delta$ as a *configuration*, and use C to range over it. D is a set of type environments representing a *decreasing set* which is a *subset of a fixed point*: a step can be taken only if Δ is in the decreasing set D. The idea is the following: since we are interested in computing all possible redexes of session environments, we avoid to further analyse the same environment twice by removing the visited environments from the (possibly infinite) set of all possible environments.

Rule Se-Top applies to configurations and checks that an environment Δ is in the decreasing set D, and Δ can move to Δ' with label α: in such case the configuration $D \diamond \Delta$ moves to the configuration $D\backslash_{\Delta} \diamond \Delta'$, where $D\backslash_{\Delta}$ notes the decreasing set D less the environment Δ.

Rule Se-Rec applies to session environments and says that $\Delta, \mathtt{p}\colon \mu X.T$ can do an internal action τ and reach the environment $\Delta, \mathtt{p}\colon T\{\mu X.T/X\}$, thus unfolding the type of the participant $\mathtt{p}$. Rule Se-Com applies to session environments and depicts a communication: when a participant $\mathtt{p}$ has a type $T_{\mathtt{p}}$ that can fire an input action $\mathtt{q}?l(v)$ and move to $T'_{\mathtt{p}}$, and a participant $\mathtt{q}$ has a type $T_{\mathtt{q}}$ that can fire an output action $\mathtt{p}!l\langle v\rangle$ and move to $T'_{\mathtt{q}}$, then $\Delta, \mathtt{p}\colon T_{\mathtt{p}}, \mathtt{q}\colon T_{\mathtt{q}}$ can fire a synchronisation action $l@\mathtt{p}\bowtie\mathtt{q}$ and move to $\Delta, \mathtt{p}\colon T'_{\mathtt{p}}, \mathtt{q}\colon T'_{\mathtt{q}}$.

Example 2. Consider the protocol introduced in § 1.1 and take $\Delta \overset{\text{def}}{=} \mathtt{s}\colon T_{\mathtt{s}}, \mathtt{c}\colon T_{\mathtt{c}}, \mathtt{a}\colon T^*_{\mathtt{a}}$. Consider a fixed point D such that $\Delta \in D$. A first application of E-Rec, Se-Rec, Se-Top allows for unfolding the type of the service $\mathtt{s}$, where we let $T^*_{\mathtt{s}} \overset{\text{def}}{=} \mathtt{c}!\mathsf{login}(\mathsf{unit}).\mathtt{a}?\mathsf{auth}(\mathsf{bool}).T_{\mathtt{s}} + \mathtt{c}!\mathsf{cancel}(\mathsf{unit}).\mathsf{end}$:

$$D \diamond \Delta \xrightarrow{\tau} D\backslash_{\Delta} \diamond \Delta, \mathtt{s}\colon T^*_{\mathtt{s}}, \mathtt{c}\colon T_{\mathtt{c}}, \mathtt{a}\colon T^*_{\mathtt{a}} \overset{\text{def}}{=} \Delta_1$$

To continue and unfold the type of the client $\mathtt{c}$, we need to verify that $\Delta_1 \in D\backslash_{\Delta}$: this follows indeed from the property of a fixed point, that is to be closed under transition, and from the fact $\Delta_1 \neq \Delta$, which holds because types are iso-recursive, and in turn $T^*_{\mathtt{s}} \neq T_{\mathtt{s}}$. We proceed as above and infer the following transition, where $T^*_{\mathtt{c}} \overset{\text{def}}{=} \mathtt{s}?\mathsf{login}(\mathsf{unit}).(\mathtt{a}!\mathsf{pwd}(\mathsf{str}).T_{\mathtt{c}} + \mathtt{a}!\mathsf{ssh}(\mathsf{unit}).T_{\mathtt{c}}) + \mathtt{s}?\mathsf{cancel}(\mathsf{unit}).\mathtt{a}!\mathsf{quit}(\mathsf{unit}).\mathsf{end}$:

$$D\backslash_{\Delta} \diamond \Delta_1 \xrightarrow{\tau} D\backslash_{\Delta,\Delta_1} \diamond \Delta, \mathtt{s}\colon T^*_{\mathtt{s}}, \mathtt{c}\colon T^*_{\mathtt{c}}, \mathtt{a}\colon T^*_{\mathtt{a}} \overset{\text{def}}{=} \Delta_2$$

Two non-deterministic transitions are available from Δ_2, and involve the synchronisation of $\mathtt{s}$ and $\mathtt{c}$: one over the label login and the other over the label cancel. The interaction below corresponds to the label login and is obtained by applying E-Out, E-Sel-L, E-In, E-Bra-L, Se-Com, Se-Top, where $T'_{\mathtt{c}} \overset{\text{def}}{=} \mathtt{a}!\mathsf{pwd}(\mathsf{str}).T_{\mathtt{c}} + \mathtt{a}!\mathsf{ssh}(\mathsf{unit}).T_{\mathtt{c}}$ and $D_2 \overset{\text{def}}{=} D\backslash_{\Delta,\Delta_1}$ and $D_3 \overset{\text{def}}{=} D_2\backslash_{\Delta_2}$:

$$D_2 \diamond \Delta_2 \xrightarrow{\mathtt{login}@\mathtt{c}\bowtie\mathtt{s}} D_3 \diamond \Delta, \mathtt{s}\colon \mathtt{a}?\mathsf{auth}(\mathsf{bool}).T_{\mathtt{s}}, \mathtt{c}\colon T'_{\mathtt{c}}, \mathtt{a}\colon T^*_{\mathtt{a}} \overset{\text{def}}{=} \Delta_3$$

The interaction over the label cancel is obtained by applying E-Out, E-Sel-R, E-In, E-Bra-R, Se-Com, Se-Top, where E-Sel-R and E-Bra-R are the right rules of E-Sel-L and E-Bra-L, respectively:

$$D_2 \diamond \Delta_2 \xrightarrow{\mathtt{cancel}@\mathtt{c}\bowtie\mathtt{s}} D_3 \diamond \Delta, \mathtt{s}\colon \mathsf{end}, \mathtt{c}\colon \mathtt{a}!\mathsf{quit}(\mathsf{unit}).\mathsf{end}, \mathtt{a}\colon T^*_{\mathtt{a}} \overset{\text{def}}{=} \Delta'_3$$

We conclude by noting that the transition system $D \diamond \Delta \xrightarrow{\alpha} D \diamond \Delta$ is indeed non-deterministic (Definition 3) by $\texttt{login@c}\bowtie\texttt{s} \neq \texttt{cancel@c}\bowtie\texttt{s}$ and $\Delta_3 \neq \Delta_3'$. □

3.2 Deterministic Session Environment Transitions

In this section, we define a deterministic LTS for environments that is the basis for the definition of *closure* in § 3.3, and in turn for the mechanisation of *compliance* (cf. § 4.1) in deductive tools of the OCaml ecosystem (cf. § 5).

The transition system $D \diamond \Delta \xrightarrow{\alpha} D' \diamond \Delta'$ is non-deterministic, for two reasons: (1) threads can reduce or interact in any order; (2) label synchronisation among two participants can occur on multiple labels and in any order.

To make the LTS deterministic (cf. Definition 3), we need four ingredients: (i) To partition the environment into minimal environments, and invoke the LTS on each minimal environment; (ii) To collect information about discarded branches and selections in synchronisations; (iii) To pass an oracle Ω that given an environment Δ returns the next two engaging participants, or the next participant firing a τ action, or nothing; (iv) To define a scheduling policy for labels of communicating participants.

We discuss (i) and (iii), and provide the signature of the deterministic LTS. Feature (i) relies on following definition; see [27] forall details.

Let $\mathsf{parties}(\mathtt{p}?l(S).T) \overset{\text{def}}{=} \mathsf{parties}(T) \cup \{\mathtt{p}\} = \mathsf{parties}(\mathtt{p}!l(S).T)$, $\mathsf{parties}(\mu X.T) \overset{\text{def}}{=} \mathsf{parties}(T)$, $\mathsf{parties}(T_1 + T_2) \overset{\text{def}}{=} \mathsf{parties}(T_1) \cup \mathsf{parties}(T_2)$, and $\mathsf{parties}(T) \overset{\text{def}}{=} \emptyset$ otherwise. Let $\mathsf{parties}(\emptyset) \overset{\text{def}}{=} \emptyset$, $\mathsf{parties}(\Delta, \mathtt{p} \colon T) \overset{\text{def}}{=} \{\mathtt{p}\} \cup \mathsf{parties}(T) \cup \mathsf{parties}(\Delta)$. Let $\Delta \backslash_{\mathsf{End}}$ project all non-ended participants of Δ.

Definition 4 (Minimal partition and environments). *A set $\{\Delta_1, \ldots, \Delta_n\} \neq \emptyset$ is a partition of $\Delta_1 \cup \cdots \cup \Delta_n$ whenever $\Delta_i \neq \emptyset$ and $\mathsf{parties}(\Delta_i) \cap \mathsf{parties}(\Delta_j) = \emptyset$ for all $\{i, j\} \subseteq \{1, \ldots, n\}$, $i \neq j$. Let $\mathcal{P}_{\mathcal{R}}(\Delta)$ be the set of all partitions of Δ. We say that Δ is minimal if there not exists $\mathcal{P}_{\mathcal{R}}(\Delta\backslash_{\mathsf{End}}) \ni S \neq \{\Delta\backslash_{\mathsf{End}}\}$ s.t. $\Delta\backslash_{\mathsf{End}} = \bigcup_{\Delta' \in S} \Delta'$. A partition $\{\Delta_1, \ldots, \Delta_n\}$ of Δ is minimal, denoted as $\mathsf{minPartition}_{\Delta}(\Delta_1, \ldots, \Delta_n)$, whenever Δ_i is minimal, for all $i \in \{1, \ldots, n\}$.*

The aim of invoking the LTS on minimal environments is to avoid the non-determinism coming from sub-systems executing unrelated behaviours. The fixed point mechanism based on decreasing sets assumes that once we re-encounter the same environment twice, we can stop since we already explored all possible computations. This is no longer sound if the system contain unrelated sub-systems. For instance, if an environment contains two participants $\mathtt{p}$ and $\mathtt{q}$ communicating with each other and reaching a fixed point after few steps, and *also* two participants $\mathtt{r}$ and $\mathtt{s}$ communicating with each other, then, depending on the oracle (see (iii)), it might be the case that the computation finishes without analysing $\mathtt{r}$ and $\mathtt{s}$ (cf. [28]). On contrast, if we consider a minimal environment, all parties are properly parsed, because the oracle is forced to analyse all sub-processes of the interacting participants. As we shall see in § 4, the minimality assumption does not pose any limitation because we perform the compliance analysis on all environments of a minimal partition.

Feature (iii) is implemented by adding a *fair* oracle returning participants willing to reduce or communicate when this option is available. The top level participant of a well-formed type T, denoted $\mathtt{top}(T)$, is a partial function indicating the unguarded participant of a branching or of a selection:

$$\mathtt{top}(\mathtt{p}?(S).T) \overset{\text{def}}{=} \mathtt{p} \qquad \mathtt{top}(\mathtt{p}!(S).T) \overset{\text{def}}{=} \mathtt{p} \qquad \mathtt{top}(T_1 + T_2) \overset{\text{def}}{=} \mathtt{top}(T_1)$$

Definition 5 (Oracle fairness). *A oracle Ω is fair whenever:*

1. $\Omega(\Delta) = (\mathtt{p}, \mathtt{q})$ *implies* $\mathtt{top}(\Delta(\mathtt{p})) = \mathtt{q}$ *and* $\mathtt{top}(\Delta(\mathtt{q})) = \mathtt{p}$
2. $\Omega(\Delta) = \mathtt{p}$ *implies* $\Delta(\mathtt{p}) = \mu X.T$
3. $\Omega(\Delta)$ *undefined implies*
   (a) *forall* $\mathtt{p} \in \text{dom}(\Delta)$ *we have* $\Delta(\mathtt{p}) \neq \mu X.T$
   (b) *there not exists* $\{\mathtt{p}, \mathtt{q}\} \subseteq \text{dom}(\Delta)$ *s.t.* $\mathtt{top}(\Delta(\mathtt{p})) = \mathtt{q}$ *and* $\mathtt{top}(\Delta(\mathtt{q})) = \mathtt{p}$

Deterministic transitions of session environments have the following form:

$$\Omega \triangleright D \diamond \Delta \xrightarrow{\alpha}_d D \diamond \Delta \blacktriangleright \Delta$$

where Δ is minimal (i), Ω is a fair oracle (iii), we assume a label scheduling policy (iv), α is a synchronisation label $l@\mathtt{p}\bowtie\mathtt{q}$ or a τ action decorated with the originating participant, denoted $\tau_{\mathtt{p}}$, and Δ after the symbol $\blacktriangleright$ is called the *sum continuation* and is a type environment or an environment placeholder, denoted ∇° (ii). We note that, w.r.t. to Definition 3, we have that $\widetilde{A} = \Omega$, the set of states $\mathcal{S}_1$ contains $D \diamond \Delta$, and the set of states $\mathcal{S}_2$ contains $D \diamond \Delta \blacktriangleright \Delta$. Moreover, $\longrightarrow_d$ is a partial function: $\Omega \triangleright D \diamond \Delta \xrightarrow{\alpha'}_d D' \diamond \Delta_1' \blacktriangleright \Delta_2'$ and $\Omega \triangleright D \diamond \Delta \xrightarrow{\alpha''}_d D'' \diamond \Delta_1'' \blacktriangleright \Delta_2''$ imply $\alpha' = \alpha''$, and $D' = D''$, $\Delta_i' = \Delta_i''$, $i = 1, 2$.

Example 3. Consider $D \diamond \Delta$ defined in Example 2. We note that Δ is minimal. Take a fair oracle Ω, and assume that the scheduling of labels follows the *lexicographic order*. First, we note that $\Omega(\Delta)$ undefined gives rise to a contradiction, because e.g. $\Delta(\mathtt{s}) = \mu X.T$. Depending on the oracle Ω, we may have $\Omega(\Delta) = \mathtt{s}$ or $\Omega(\Delta) = \mathtt{c}$, because any other combination would contradict Definition 5.

Assume $\Omega(\Delta) = \mathtt{s}$. A first step let us infer the reduction of the service, where $\Delta_1 \overset{\text{def}}{=} \Delta, \mathtt{s}\colon T_{\mathtt{s}}^*, \mathtt{c}\colon T_{\mathtt{c}}, \mathtt{a}\colon T_{\mathtt{a}}^*$, and $\mathsf{minimal}(\Delta_1)$.

$$\Omega \triangleright D \diamond \Delta \xrightarrow{\tau_{\mathtt{s}}}_d D\backslash_{\Delta} \diamond \Delta_1 \blacktriangleright \nabla^\circ$$

Next, we assume that $\Omega(\Delta_1) = \mathtt{c}$, where $\Delta_2 \overset{\text{def}}{=} \Delta, \mathtt{s}\colon T_{\mathtt{s}}^*, \mathtt{c}\colon T_{\mathtt{c}}^*, \mathtt{a}\colon T_{\mathtt{a}}^*$.

$$\Omega \triangleright D\backslash_{\Delta} \diamond \Delta_1 \xrightarrow{\tau_{\mathtt{c}}}_d D\backslash_{\Delta,\Delta_1} \diamond \Delta_2 \blacktriangleright \nabla^\circ$$

In the next round we have $\mathsf{minimal}(\Delta_2)$ and $\Omega(\Delta_2) = (\mathtt{c}, \mathtt{s})$, and the algorithm picks the first label in the intersection of the labels of $\mathtt{c}$ and $\mathtt{s}$, that is cancel:

$$\Omega \triangleright D_2 \diamond \Delta_2 \xrightarrow{\mathtt{cancel}@\mathtt{c}\bowtie\mathtt{s}}_d D_3 \diamond \Delta'' \blacktriangleright \Delta'$$

where D_2, D_3 are defined in Example 2 and

$$\begin{aligned}
\Delta'' &\overset{\text{def}}{=} \mathsf{s}\colon \mathsf{end}, \mathsf{c}\colon \mathsf{a}!\mathsf{quit}(\mathsf{unit}).\mathsf{end}, \mathsf{a}\colon T^*_{\mathsf{a}}\\
\Delta' &\overset{\text{def}}{=} \mathsf{s}\colon \mathsf{c}!\mathsf{login}(\mathsf{unit}).\mathsf{a}?\mathsf{auth}(\mathsf{bool}).T_{\mathsf{s}}, \mathsf{c}\colon \mathsf{s}?\mathsf{login}(\mathsf{unit}).T'_{\mathsf{c}}, \mathsf{a}\colon T^*_{\mathsf{a}}\\
T'_{\mathsf{c}} &\overset{\text{def}}{=} \mathsf{a}!\mathsf{pwd}(\mathsf{str}).T_{\mathsf{c}} + \mathsf{a}!\mathsf{ssh}(\mathsf{unit}).T_{\mathsf{c}}
\end{aligned}$$

After this sequence of transitions, we have two minimal environments Δ'' and Δ' corresponding to the redex of the interaction of the service s and the client c over the label cancel, and to the environment prompt to let s and c interact over the label login, respectively. The idea is to deterministically visit all the binary trees spawned by further transitions starting from $D_3 \diamond \Delta''$ and from $D_3 \diamond \Delta'$, respectively, as we discuss in the next section. □

3.3 Closure

The aim of the deterministic LTS presented in § 3.2 is to be used by the function that computes the *compliance* of session environments in the typing system (cf. § 4). Compliance analyses all final environments computed by the closure of the deterministic transitions originating from a type environment.

More specifically, we consider the *semireflexive-transitive closure* of the deterministic lts $\longrightarrow_d$, denoted $\Longrightarrow$. Semireflexivity means that a configuration is related with itself only if is stuck, that is it cannot fire any transition.

We are interested in applying closure to environments preserving minimality.

Definition 6 (Stuck environment). *A minimal environment Δ is stuck w.r.t. an oracle Ω and a decreasing set D, denoted $\mathtt{stuck}_{\Omega,D}(\Delta)$, if there not exists α, Δ_1, Δ_2 such that $\Omega \triangleright D \diamond \Delta \overset{\alpha}{\longrightarrow}_d D' \diamond \Delta_1 \blacktriangleright \Delta_2$.*

Definition 7 (Closure). *Define:*

$$\text{C-RFL}\,\frac{\mathtt{stuck}_{\Omega,D}(\Delta)}{\Omega \triangleright D \diamond \Delta \Longrightarrow D \diamond \Delta} \qquad \text{C-ERR}\,\frac{\neg\mathtt{minimal}(\Delta)}{\Omega \triangleright D \diamond \Delta \Longrightarrow \mathtt{err}}$$

$$\text{C-TRA}\,\frac{\Omega \triangleright D \diamond \Delta \overset{\alpha}{\longrightarrow}_d D' \diamond \Delta_1 \blacktriangleright \Delta_2 \quad \Omega \triangleright D' \diamond \Delta_1 \Longrightarrow \widetilde{E_1} \quad \Omega \triangleright D' \diamond \Delta_2 \Longrightarrow \widetilde{E_2}}{\Omega \triangleright D \diamond \Delta \Longrightarrow \widetilde{E_1}, \widetilde{E_2}}$$

The closure of a minimal environment Δ w.r.t. a decreasing set D s.t. $\Delta \in D$ and a fair oracle Ω is defined by the following rule:

$$\text{C-TOP}\,\frac{\Omega \triangleright D \diamond \Delta \Longrightarrow D_1 \diamond \Delta_1, \cdots, D_n \diamond \Delta_n}{\mathtt{closure}_{\Omega,D}(\Delta) = \Delta_1, \ldots, \Delta_n}$$

Given a a fair oracle Ω, the relation $\Longrightarrow$ associates a configuration C to a *non-empty* tuple of *e-configurations* $E_1, \ldots, E_n$, denoted as $\widetilde{E}$, where each E_i is a configuration C or the *failure* $\mathtt{err}$. Given a configuration $C = D \diamond \Delta$, three cases may arise. If C is stuck, that is C cannot fire any transition, then we apply rule [C-RFL] and relate C with itself, else if C is not minimal, then we we apply rule [C-ERR] and relate C with $\mathtt{err}$. Otherwise we have that C fires an action and reaches the redex $\Delta' \diamond \Delta_1 \blacktriangleright \Delta_2$: we apply rule [C-TRA] and whenever $\Delta' \diamond \Delta_1$ is related by $\Longrightarrow$ to the e-configurations $\widetilde{E_1}$, and $\Delta' \diamond \Delta_2$ is related by $\Longrightarrow$ to $\widetilde{E_2}$, we let C be related by $\Longrightarrow$ to $\widetilde{E_1}, \widetilde{E_2}$.

The $\mathtt{closure}$ of a session environment Δ is defined iff $\Longrightarrow$ does not relate Δ with failures. If this is the case, then $\Longrightarrow$ relates Δ with configurations $\widetilde{C}$: the function strips off all decreasing sets and associates Δ to a set of minimal stuck environments. It is worth noting that $\mathtt{closure}$ is a *terminating function*, because it is deterministic and it has $|D|$ as decreasing measure.

Example 4. We continue the analysis started in Example 3 and find a subset of $\mathtt{closure}_{\Omega,D}(\Delta)$, which is defined because Δ and its redexes are minimal. Remember T^*_{a} defined in § 1.1: $T^*_{\mathsf{a}} \stackrel{\text{def}}{=} \mathsf{c}?\mathsf{pwd}(\mathsf{str}).\mathsf{s}!\mathsf{auth}(\mathsf{bool}).T_{\mathsf{a}} + \mathsf{c}?\mathsf{ssh}(\mathsf{unit}).\mathsf{s}!\mathsf{auth}(\mathsf{bool}).T_{\mathsf{a}} + \mathsf{c}?\mathsf{quit}(\mathsf{unit}).\mathsf{end}$. Consider $D_3 \diamond \Delta'' \blacktriangleright \Delta'$ defined in Example 3:

$$\Omega \triangleright D \diamond \Delta \xrightarrow{\tau_{\mathsf{s}}}_d \xrightarrow{\tau_{\mathsf{c}}}_d \xrightarrow{\mathtt{cancel@c{\bowtie}s}}_d D_3 \diamond \Delta'' \blacktriangleright \Delta'$$

To calculate the closure of Δ w.r.t. D, we need to analyse the closures of Δ'' and Δ' w.r.t. D_3, respectively. We have that Δ'' and Δ' are minimal: we analyse the former closure, and note that $D_3 \diamond \Delta''$ is not stuck, i.e. the client c and the server a can communicate on quit. Assume $\Omega(\Delta'') = (\mathsf{a}, \mathsf{c})$. We have:

$$\textsc{C-Tra}\,\frac{\Omega \triangleright D_3 \diamond \Delta'' \xrightarrow{\mathtt{quit@a{\bowtie}c}}_d D_3\backslash_{\Delta''} \diamond \mathsf{s}\colon \mathsf{end}, \mathsf{c}\colon \mathsf{end}, \mathsf{a}\colon \mathsf{end} \blacktriangleright \nabla^\circ \qquad (A)}{\Omega \triangleright D_3 \diamond \Delta'' \Longrightarrow D_3\backslash_{\Delta''} \diamond \mathsf{s}\colon \mathsf{end}, \mathsf{c}\colon \mathsf{end}, \mathsf{a}\colon \mathsf{end}}$$

$$(A)\ \textsc{C-Rfl}\,\frac{}{\Omega \triangleright D_3\backslash_{\Delta''} \diamond \mathsf{s}\colon \mathsf{end}, \mathsf{c}\colon \mathsf{end}, \mathsf{a}\colon \mathsf{end} \Longrightarrow D_3\backslash_{\Delta''} \diamond \mathsf{s}\colon \mathsf{end}, \mathsf{c}\colon \mathsf{end}, \mathsf{a}\colon \mathsf{end}}$$

We can thus infer $(\mathsf{s}\colon \mathsf{end}, \mathsf{c}\colon \mathsf{end}, \mathsf{a}\colon \mathsf{end}) \in \mathtt{closure}_{\Omega,D}(\Delta)$. $\square$

4 Iso-Recursive Multiparty Type System

The typing rules for processes and sessions are defined in Figure 3; we refer to the technical report [27] for the rules for expressions.

Typing judgements for processes have the form $\Gamma \vdash P\colon T$, where Γ maps variables to sorts and process variables to types:

$$\Gamma := \emptyset \mid \Gamma, x\colon S \mid \Gamma, \chi\colon T$$

Typing judgements for sessions have the form $\Gamma \Vdash \mathcal{M}\colon \Delta$, where Δ is the session environment introduced in § 3, that is a map from participants to types, and invoke the type system $\vdash$. The type system for sessions $\Vdash$ only invokes the type system for processes $\vdash$ with well-formed types (cf. § 2): for this reason, the typing rules for processes involving type sums can be simplified (cf. rules T-Sum,T-Sum-L,T-Sum-R).

The rule depicting the essence of iso-recursive multiparty session types is T-Rec. In order to allow Γ to type a recursion process $\mu\chi.P$ with a type $\mu X.T$, it must be the case that $\Gamma, \chi\colon \mu X.T$ types the continuation P with the unfolded type $T\{\mu X.T/X\}$. That is, in our iso-recursive setting the continuation must be typed by explicitly unfolding the recursive type. This is different from the equi-recursive approach, e.g. [23], where the type of $\mu\chi.P$ and the type of the

Sorting rules: $\boxed{\Gamma \vdash e \colon S}$

Typing rules for processes: $\boxed{\Gamma \vdash P \colon T}$

$$\text{T-End}\,\frac{}{\Gamma \vdash 0 \colon \mathsf{end}} \qquad \text{T-Rec}\,\frac{\Gamma, \chi \colon \mu X.T \vdash P \colon T\{\mu X.T/X\}}{\Gamma \vdash \mu\chi.P \colon \mu X.T}$$

$$\text{T-Var}\,\frac{\Gamma(\chi) = \mu X.T}{\Gamma \vdash \chi \colon \mu X.T} \qquad \text{T-Inp}\,\frac{\Gamma, x \colon S \vdash P \colon T}{\Gamma \vdash \mathtt{r}?l(x).P \colon \mathtt{r}?l(S).T}$$

$$\text{T-Out}\,\frac{\Gamma \vdash e \colon S \qquad \Gamma \vdash P \colon T}{\Gamma \vdash \mathtt{r}!l\langle e\rangle.P \colon \mathtt{r}!l(S).T} \qquad \text{T-Sum}\,\frac{\Gamma \vdash P \colon T_1 \qquad \Gamma \vdash Q \colon T_2}{\Gamma \vdash P + Q \colon T_1 + T_2}$$

$$\text{T-Sum-L}\,\frac{\Gamma \vdash P \colon T_1}{\Gamma \vdash P \colon T_1 + T_2} \qquad \text{T-Sum-R}\,\frac{\Gamma \vdash P \colon T_2}{\Gamma \vdash P \colon T_1 + T_2}$$

$$\text{T-If}\,\frac{\Gamma \vdash e \colon \mathsf{bool} \qquad \Gamma \vdash P \colon T \qquad \Gamma \vdash Q \colon T}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \colon T}$$

Typing rules for sessions: $\boxed{\Gamma \Vdash \mathcal{M} \colon \Delta}$

$$\text{T-Thr}\,\frac{\Gamma \vdash P \colon T \qquad \mathrm{WF}(T)}{\Gamma \Vdash \mathtt{p} \lhd P \colon \mathtt{p} \colon T}$$

$$\text{T-Ses}\,\frac{\begin{array}{c}\Gamma \Vdash \mathtt{p}_1 \lhd P_1 \colon \mathtt{p}_1 \colon T_1 \quad \cdots \quad \Gamma \Vdash \mathtt{p}_n \lhd P_n \colon \mathtt{p}_n \colon T_n \quad \Delta = \mathtt{p}_1 \colon T_1, \ldots, \mathtt{p}_n \colon T_n \\ \mathsf{minPartition}_\Delta(\Delta_1, \ldots, \Delta_k) \qquad \forall j \in \{1, \ldots, k\} \,.\, \mathsf{comp}(\Delta_j)\end{array}}{\Gamma \Vdash \parallel_{i \in \{1,..,n\}} \mathtt{p}_i \lhd P_i \colon \Delta}$$

Fig. 3. Type system

continuation P can be equal, because types $\mu X.T$ and $T\{\mu X.T/X\}$ are equal. For the same reason, in rule T-Var an environment $\Gamma, \chi \colon \mu X.T$ assigns the type $\mu X.T$ to the process variable χ: note that it is not possible to assign a non-recursive type to process variables.

Rule T-Inp allows Γ to type a input process $\mathtt{r}?l(x).P$ with type $\mathtt{r}?l(S).T$ whenever $\Gamma, x \colon S$ assigns the type T to the continuation P. Dually, rule T-Out allows Γ to type an output process $\mathtt{r}!l\langle e\rangle.P$ with type $\mathtt{r}?l(S).T$ whenever the expression has sort S and Γ assigns the type T to the continuation P.

Rule T-Sum is used for branching and selection, that are sums containing only input types from the same participant and without duplicated labels, or output types from the same participant and without duplicated labels, respectively (cf. Well-Formed Types in § 2, and Definition 2). Note indeed that well-formed types do not contain types of the form e.g. $T_1 + \mu X.T_2$, or $\mathsf{end} + T$. The rule says that if Γ can be used to type a process P_1 with type T_1, and a process P_2 with type T_2, then Γ types $P_1 + P_2$ with type $T_1 + T_2$.

While rule T-Sum types exactly each input and output with their corresponding input and output type singletons, rule T-Sum-L allows for typing a process P having type T_1 with the type $T_1 + T_2$. For instance, if P is the branch-

$$P_{\mathtt{a}} \overset{\text{def}}{=} \mu\chi.(P_1 + P_2) \quad P_1 \overset{\text{def}}{=} \mathtt{c}?\mathsf{pwd}(x).\mathit{Check}_{\mathtt{a}}$$
$$P_2 \overset{\text{def}}{=} \mathtt{c}?\mathsf{ssh}(x).\mathtt{s}!\mathsf{auth}\langle\mathsf{true}\rangle.\chi + \mathtt{c}?\mathsf{quit}(x).\mathbf{0}$$
$$\mathit{Check}_{\mathtt{a}} \overset{\text{def}}{=} \mathsf{if}\, x = \text{“miau”}\, \mathsf{then}\, \mathtt{s}!\mathsf{auth}\langle\mathsf{true}\rangle.\chi\, \mathsf{else}\, \mathtt{s}!\mathsf{fail}\langle\rangle.\mathbf{0}$$
$$T' \overset{\text{def}}{=} \mathtt{c}?\mathtt{pwd}(\mathsf{str}).(\mathtt{s}!\mathsf{auth}(\mathsf{bool}).X + \mathtt{s}!\mathsf{fail}(\mathsf{unit}).\mathsf{end})$$
$$T'' \overset{\text{def}}{=} \mathtt{c}?\mathsf{ssh}(\mathsf{unit}).\mathtt{s}!\mathsf{auth}(\mathsf{bool}).X + \mathtt{c}?\mathsf{quit}(\mathsf{unit}).\mathsf{end} \quad T \overset{\text{def}}{=} T' + T''$$

Fig. 4. Variant of authorisation server in § 1.1

ing process $\mathtt{r}?l_1(x).P_1 + \cdots + \mathtt{r}?l_n(x).P_n$ then we can use T-Sum-L to assign to P the type $\&_{i\in\{1,\ldots,n+1\}}\mathtt{r}?l_i(S_i).T_i$. Rule T-Sum-R does the same thing, on the right: if P has type T_2 then we can use the rule to assign to P the type $T_1 + T_2$.

The increased flexibility offered by rules T-Sum-L, T-Sum-R is used in the rule for if-then-else, that is T-If. In order to type process $\mathsf{if}\, e\, \mathsf{then}\, P\, \mathsf{else}\, Q$ with type T we require that e has a boolean sort, and that both P and Q have type T. To allow P and Q to use different labels to communicate in input/output with a participant, we use rules T-Sum-L and T-Sum-R in the premises of T-If, thus mimicking a simple form of subtyping. The next example illustrates this idea.

Example 5. Consider the variant of Figure 4 of the authorisation server $\mathtt{a}$ in § 1.1 such that $\mathtt{a}$ verifies the password sent by the client $\mathtt{c}$ while allowing only one attempt: if the password is wrong, $\mathtt{a}$ sends fail to the service $\mathtt{s}$ and stops. We informally discuss the typing of the authorisation server $P_{\mathtt{a}}$, and omit the types of the other participants. A formal derivation is included in [27].

Let $\Gamma \overset{\text{def}}{=} \chi\colon \mu X.T, x\colon \mathsf{str}$, consider the two branches of $\mathit{Check}_{\mathtt{a}}$, and let $T_{\text{if}} \overset{\text{def}}{=} \mathtt{s}!\mathsf{auth}(\mathsf{bool}).\mu X.T + \mathtt{s}!\mathsf{fail}(\mathsf{unit}).\mathsf{end}$. The left branch $\mathtt{s}!\mathsf{auth}\langle\mathsf{true}\rangle.\chi$ can be assigned to T_{if} under Γ by using T-Sum-L, T-Out, T-Var. The right branch $\mathtt{s}!\mathsf{fail}\langle\rangle.\mathbf{0}$ can be assigned to T_{if} under Γ by using T-Sum-R, T-Out, T-End. By applying T-If we thus assign T_{if} to $\mathit{Check}_{\mathtt{a}}$ under Γ ; in turn, process P_1 is assigned to $T_1 \overset{\text{def}}{=} \mathtt{c}?\mathtt{pwd}(\mathsf{str}).T_{\text{if}}$ under $\chi\colon \mu X.T$ by using T-Inp. We note that $T_1 = T'\{\mu X.T/X\}$. Process P_2 is assigned to $T_2 \overset{\text{def}}{=} \mathtt{c}?\mathsf{ssh}(\mathsf{unit}).\mathtt{s}!\mathsf{auth}(\mathsf{bool}).\mu X.T + \mathtt{c}?\mathtt{quit}(\mathsf{unit}).\mathsf{end}$ under $\chi\colon \mu X.T$: we omit all details. We note that $T_2 = T''\{\mu X.T/X\}$.

We use T-Sum to assign $T'\{\mu X.T/X\} + T''\{\mu X.T/X\} = T\{\mu X.T/X\}$ to $P_1 + P_2$ under $\chi\colon \mu X.T$. We conclude by using T-Rec to assign $\mu X.T$ to $P_{\mathtt{a}}$ under the empty environment, thus typing the authorisation server. □

Type checking sessions. The typing rules for sessions of Figure 3 have the form $\Gamma \Vdash \mathcal{M}\colon \Delta$ and use the rules for processes $\Gamma \vdash P\colon T$. The system relies on the notion of *minimal partition* (cf. Definition 4).

Rule T-Thr is used for single threads and says that if the type system for processes $\vdash$ can be used to type a process P with a well-formed type T (cf. § 2), then the type system $\Vdash$ assigns the typing $\mathtt{p}\colon T$ to the thread $\mathtt{p} \lhd P$.

Rule T-Ses is the top-level rule used to type-check the multiparty session. In order to type-check a session composing the threads $\mathtt{p}_1 \lhd P_1, \ldots, \mathtt{p}_n \lhd P_n$ with the session environment $\Delta = \mathtt{p}_1\colon T_1, \ldots, \mathtt{p}_n\colon T_n$, we require two things:

1. Each thread $\mathtt{p}_i \triangleleft P_i$ is typed with the environment $\mathtt{p}_i : T_i$, for $i = 1, \dots n$;
2. Each environment Δ_j of the minimal partition $\{\Delta_1, \dots, \Delta_k\}$ of Δ satisfies *compliance*, denoted $\mathsf{comp}(\Delta_j)$.

Compliance resembles the approach based on safe contexts (e.g. [49, Definition 4.1]), although is fully computational.

4.1 Compliance

Intuitively, a session typed by a compliant environment never reaches an *error*, that is a deadlocked system, or a redex containing two participants $\mathtt{p}$ and $\mathtt{q}$ that are willing to communicate, e.g. $\mathtt{p}$ is sending an output to $\mathtt{q}$, and $\mathtt{q}$ is receiving an input from $\mathtt{p}$, or vice-versa, but they mismatch the communication label and/or the type payload, or both $\mathtt{p}$ and $\mathtt{q}$ are sending (receiving) a value to each other: that is, there is a mismatch that makes the two participants stuck.

The formal definition of compliance relies on the *closure* of $\longrightarrow_d$ introduced in § 3, and of the formal definition of error below. Let the *tagged labels* of a type T, denoted $\mathcal{L}(T)$, be defined inductively as follows: $\mathcal{L}(\mathtt{r}!l(S).T) \overset{\text{def}}{=} \{l@S\}$, $\mathcal{L}(\mathtt{r}?l(S).T) \overset{\text{def}}{=} \{l@S\}$, $\mathcal{L}(T_1 + T_2) \overset{\text{def}}{=} \mathcal{L}(T_1) \cup \mathcal{L}(T_2)$, $\mathcal{L}(T) \overset{\text{def}}{=} \emptyset$ otherwise.

Definition 8 (Well-formed environment). *A session environment Δ is well-formed, denoted $WF(\Delta)$, whenever $\mathtt{p} \in \mathrm{dom}(\Delta)$ implies $WF(\Delta(\mathtt{p}))$.*

Definition 9 (Communication mismatch). *A well-formed session environment Δ is a communication mismatch whenever there exists $\{\mathtt{p}, \mathtt{q}\} \subseteq \mathrm{dom}(\Delta)$ such that one of the following cases arise:*

$$\begin{array}{lll}
\Delta(\mathtt{p}) = \oplus_{i\in I}\mathtt{q}!l_i(S_i).T_i & \Delta(\mathtt{q}) = \oplus_{j\in J}\mathtt{p}!l_j(S_j).T_j & \\
\Delta(\mathtt{p}) = \&_{i\in I}\mathtt{q}?l_i(S_i).T_i & \Delta(\mathtt{q}) = \&_{j\in J}\mathtt{p}?l_j(S_j).T_j & \\
\Delta(\mathtt{p}) = \oplus_{i\in I}\mathtt{q}!l_i(S_i).T_i & \Delta(\mathtt{q}) = \&_{j\in J}\mathtt{p}?l_j(S_j).T_j & \mathcal{L}(\Delta(\mathtt{p})) \cap \mathcal{L}(\Delta(\mathtt{q})) = \emptyset \\
\Delta(\mathtt{p}) = \&_{i\in I}\mathtt{q}?l_i(S_i).T_i & \Delta(\mathtt{q}) = \oplus_{j\in J}\mathtt{p}!l_j(S_j).T_j & \mathcal{L}(\Delta(\mathtt{p})) \cap \mathcal{L}(\Delta(\mathtt{q})) = \emptyset
\end{array}$$

The notion of deadlock is insensitive to decreasing sets and determinism, and is based on the non-deterministic transition system $\Delta \xrightarrow{\alpha} \Delta$ of Figure 2.

Definition 10 (Deadlock). *Let $\mathsf{consumed}(\Delta) \overset{def}{=} \forall \mathtt{p} \in \mathrm{dom}(\Delta) \,.\, \Delta(\mathtt{p}) = \mathsf{end}$. A session environment Δ is a deadlock when both (1) there not exists α, Δ' such that $\Delta \xrightarrow{\alpha} \Delta'$, and (2) $\neg\mathsf{consumed}(\Delta)$.*

Definition 11 (Error). *A well-formed environment Δ is an error whenever Δ is a communication mismatch, or Δ is a deadlock.*

Definition 12 (Compliance). *Let Δ be a minimal well-formed environment. Define $\mathsf{comp}(\Delta)$ whenever for all fair oracles Ω and fixed points D including Δ, if $\mathtt{closure}_{\Omega,D}(\Delta) = \Delta_1, \dots, \Delta_n$ then Δ_i is not an error, for all $i \in \{1, \dots, n\}$.*

Example 6. Consider the minimal well-formed environment Δ'' introduced at the end of § 1.1, and the claim $\neg\mathsf{comp}(\Delta'')$, which follows from Δ'' reaching the deadlocked environment $\Delta_{\texttt{lock}} = \mathtt{s}\colon \mathtt{a}?\mathsf{auth}(\mathsf{bool}).T_{\mathtt{s}},\ \mathtt{c}\colon \mathtt{a}!\mathsf{pwd}(\mathsf{str}).T_{\mathtt{c}} + \mathtt{a}!\mathsf{ssh}(\mathsf{unit}).T_{\mathtt{c}}$, $\mathtt{a}\colon \mathsf{end}$. We prove the claim by using a Lemma mapping non-deterministic transitions to deterministic transitions. We start by a sequence of (non-deterministic) transitions from Δ'' that lead to $\Delta_{\texttt{lock}}$, and use the result to find a fair oracle Ω mimicking the sequence:

$$D \diamond \Delta'' \xrightarrow{\tau_{\mathtt{s}}} \xrightarrow{\tau_{\mathtt{c}}} \xrightarrow{\texttt{login}@\mathtt{c}\bowtie\mathtt{s}} \xrightarrow{\texttt{ssh}@\mathtt{a}\bowtie\mathtt{c}} \xrightarrow{\texttt{auth}@\mathtt{s}\bowtie\mathtt{a}} \tag{1}$$

$$D_1 \diamond \mathtt{s}\colon T_{\mathtt{s}}, \mathtt{c}\colon T_{\mathtt{c}}, \mathtt{a}\colon T'_{\mathtt{a}} \xrightarrow{\tau_{\mathtt{s}}} \xrightarrow{\tau_{\mathtt{c}}} \xrightarrow{\texttt{login}@\mathtt{c}\bowtie\mathtt{s}} \xrightarrow{\tau_{\mathtt{a}}} \xrightarrow{\texttt{ssh}@\mathtt{a}\bowtie\mathtt{c}} \xrightarrow{\texttt{auth}@\mathtt{s}\bowtie\mathtt{a}} \tag{2}$$

$$D_2 \diamond \mathtt{s}\colon T_{\mathtt{s}}, \mathtt{c}\colon T_{\mathtt{c}}, \mathtt{a}\colon \mathsf{end} \xrightarrow{\tau_{\mathtt{s}}} \xrightarrow{\tau_{\mathtt{c}}} \xrightarrow{\texttt{login}@\mathtt{c}\bowtie\mathtt{s}} D_3 \diamond \Delta_{\texttt{lock}} \tag{3}$$

The transitions in (1) correspond to a first round of the protocol, which leads the service $\mathtt{s}$ and the client $\mathtt{c}$ to re-initialise, while the authorisation server $\mathtt{a}$ reaches the type $T'_{\mathtt{a}} = \mu X.(\mathtt{c}?\texttt{pwd}(\mathsf{str}).\mathtt{s}!\mathsf{auth}(\mathsf{bool}).X + \mathtt{c}?\mathsf{ssh}(\mathsf{unit}).\mathtt{s}!\mathsf{auth}(\mathsf{bool}).\mathsf{end}$ $+\mathtt{c}?\texttt{quit}(\mathsf{unit}).\mathsf{end})$. The transitions in (2) correspond to a second round of the protocol, which leads the service $\mathtt{s}$ and the client $\mathtt{c}$ to re-initialise, while the authorisation server $\mathtt{a}$ reaches the type end. The transitions in (3) correspond to the starting of the protocol where the service $\mathtt{s}$ sends a login request to the client $\mathtt{c}$. After that, both the service and the client waits to interact with the server $\mathtt{a}$, which has ended. Note that $\texttt{stuck}_{\Omega,D_3}(\Delta_{\texttt{lock}})$.
We apply a multi-step Lemma (see [27]) and infer $\Omega \triangleright D \diamond \Delta'' \implies \widetilde{C_1}, D_3 \diamond \Delta_{\texttt{lock}}, \widetilde{C_2}$. By Definition 7, we have $\Delta_{\texttt{lock}} \in \mathsf{closure}_{\Omega,D}(\Delta'')$. To prove $\neg\mathsf{comp}(\Delta'')$, we show that $\Delta_{\texttt{lock}}$ is an error. In fact, $\Delta_{\texttt{lock}}$ is a deadlock (cf. Definition 10), because it cannot fire any action, and because there is a participant that has not finished, e.g. $\Delta_{\texttt{lock}}(\mathtt{s}) \neq \mathsf{end}$. By Definition 11, $\Delta_{\texttt{lock}}$ is an error. □

Example 7. Consider the minimal well-formed environment Δ of the authorisation protocol in § 1.1. We claim that for any fair oracle Ω and fixed point $D \ni \Delta$, the closure of Δ returns two environments, where $\Delta^{\mathsf{end}} \overset{\text{def}}{=} \mathtt{s}\colon \mathsf{end}, \mathtt{c}\colon \mathsf{end}, \mathtt{a}\colon \mathsf{end}$: $\mathsf{closure}_{\Omega,D}(\Delta) = \{\Delta, \Delta^{\mathsf{end}}\}$. Following this claim, we have $\mathsf{comp}(\Delta)$. In fact, both Δ and Δ^{end} are not errors. By definition, neither Δ nor Δ^{end} is a mismatch: the latter case is clear; in the former case, the unique unguarded sum of prefixes is the branching of the authorisation service $\mathtt{a}$ below, while the type of $\mathtt{c}$ is guarded:

$$\mathtt{c}?\texttt{password}(\mathsf{str}).\mathtt{s}!\mathsf{auth}(\mathsf{bool}).T_{\mathtt{a}} + \mathtt{c}?\mathsf{ssh}(\mathsf{unit}).\mathtt{s}!\mathsf{auth}(\mathsf{bool}).T_{\mathtt{a}} + \mathtt{c}?\texttt{quit}(\mathsf{unit}).\mathsf{end}$$

Moreover, neither Δ nor Δ^{end} is a deadlock. Δ can indeed take a step: the environment is in the closure because it is first contained in the initial decreasing set D and then re-encountered after a sequence of interactions. The claim can be verified by using the certified implementation in § 5. □

Remark 1. In [49] an environment is deadlock-free if for all redexes Γ reachable in multiple steps we have that if Γ does not move then its range contains only the type end. Conversely, Definition 10 expresses a negative property, and in turn we transform the implication $\mathsf{stuck}(\Gamma) \rightarrow \mathsf{consumed}(\Gamma)$ of [49] into its negation: $\mathsf{stuck}(\Gamma) \wedge \neg\mathsf{consumed}(\Gamma)$. □

4.2 Subject Reduction and Progress

We conclude this section by showing that the typing system satisfies subject reduction and progress. We outline the sketch of the proof of subject reduction, and refer to [27] for all details, including the proof of progress.

The purpose of the subject reduction theorem is to establish that if a session $\mathcal{M}$ is well-typed and does a step α and reaches the session $\mathcal{M}'$, then $\mathcal{M}'$ is well-typed. Assume that $\Gamma \Vdash \mathcal{M} \colon \Delta$. To assess subject reduction, we provide an environment Δ' s.t. $\Gamma \Vdash \mathcal{M}' \colon \Delta'$. Since the step $\mathcal{M} \xrightarrow{\alpha} \mathcal{M}'$ is non-deterministic, we match this step with a non-deterministic environment transition (cf. Figure 2).

A key result to establish subject reduction is that compliance (cf. Definition 12) is preserved by non-deterministic transitions of session environments.

Lemma 1. *Let Δ be minimal. If $D \diamond \Delta \xrightarrow{\alpha} D' \diamond \Delta'$ then there exists a fair oracle Ω and environment Δ'' s.t. $\Omega \triangleright D \diamond \Delta \xrightarrow{\alpha}_d D' \diamond \Delta' \blacktriangleright \Delta''$.*

Lemma 2. *If $\mathsf{comp}(\Delta)$ and $\Omega \triangleright D \diamond \Delta \xrightarrow{\alpha}_d D' \diamond \Delta' \blacktriangleright \Delta''$ then $\mathsf{comp}(\Delta')$.*

Corollary 1 (Compliance preservation). *If $\mathsf{comp}(\Delta)$ and $D \diamond \Delta \xrightarrow{\alpha} D' \diamond \Delta'$ then $\mathsf{comp}(\Delta')$.*

Lemma 3. *If $\Gamma \Vdash \mathcal{M}_1 \colon \Delta$ and $\mathcal{M}_1 \equiv \mathcal{M}_2$ then $\Gamma \Vdash \mathcal{M}_2 \colon \Delta$.*

Proof. If follows from the inversion of $\Gamma \Vdash \mathcal{M}_1 \colon \Delta$ and the definition of $\equiv$. The result is mechanised in Coq. □

Theorem 1 (Subject Reduction). *Let $\mathcal{M}$ be a closed session, and let D be a fixed point including Δ. Assume (1) $\Gamma \Vdash \mathcal{M} \colon \Delta$ and (2) $\mathcal{M} \xrightarrow{\alpha} \mathcal{M}'$. We have $\Gamma \Vdash \mathcal{M}' \colon \Delta$ or $D \diamond \Delta \xrightarrow{\alpha} D' \diamond \Delta'$ and $\Gamma \Vdash \mathcal{M}' \colon \Delta'$.*

Proof. By induction on (2), using value and process substitution (mechanised in Coq), Lemma 3, and Corollary 1. □

Let $\mathsf{Ended}(\,\|_{i\in I}\ \mathtt{p}_i \triangleleft P_i)$ when for all $i \in I$ we have $P_i = \mathbf{0}$.

Theorem 2 (Progress). *Let $\mathcal{M}$ be a closed session. If $\Gamma \Vdash \mathcal{M} \colon \Delta$ and does not exist $\mathcal{M}'$ s.t. $\mathcal{M} \xrightarrow{\tau} \mathcal{M}'$ or $\mathcal{M} \xrightarrow{l@\mathtt{p}\bowtie\mathtt{q}} \mathcal{M}'$, for all $l, \mathtt{p}, \mathtt{q}$, then $\mathsf{Ended}(\mathcal{M})$.*

5 Automated Deductive Verification of Compliance

The typing system presented in § 4 relies on the notion of *compliance*, which is defined theoretically by relying on the novel definitions of *deterministic session environment transitions* and *closure* introduced in § 3. In this section, we showcase how these theoretical notions can de deployed *soundly* in mainstream programming languages and compilers by presenting a *reference implementation* *of compliance* and by mechanising the properties of the implementation, which are that compliant environment are mismatch-free and deadlock-free.

Our goal is to define *compliance* as a computable function that decides when a session environment has a "good behaviour", and in turn can be assigned by the typing system to a session. We note that computability is an essential prerequisite for decidable type checking while assigning non-compliant environments to sessions is unsound because it invalidates progress, and must be avoided.

Towards this aim, we need to (1) deploy the function; (2) provide a mechanised proof that the function terminates; (3) provide a mechanised proof that the function decides freedom from mismatches and deadlocks. This result is established once (by the type system designer): after that, the function can be used each time we invoke the type checker on a session process.

The proofs and their mechanisation in (2) and (3) are necessary because the designer can deploy a wrong implementation, e.g. it could have forgotten a case leading to an environment deadlock, thus allowing to type check sessions that deadlock at runtime. By providing a computer-assisted proof that the implementation rules out errors and deadlocks in environments, we can rely on Theorem 2 to obtain that sessions typed by accepted environments do not deadlock.

In the remainder of the section, we tackle the requirements (1), (2) and (3) by defining function *compliance* and its *behavioural specification*, that is the contract of the function [40]. We choose OCaml as target language and use tools of the OCaml ecosystem relying on Why3 [19] to enable automated deductive verification of behavioural specifications by using constraint solvers, e.g. [16,42,4], while supporting imperative features, ghost code [18], and interactive proofs.

The verification has been done by using Cameleer [45,44], which in turn relies on [9,19]. Proofs of lemmas requiring induction are done interactively in Why3.

5.1 Structure of the Implementation

To implement *closure* (cf. Definition 7) in OCaml, we use function `cstep` receiving a fair oracle Ω, a decreasing set D, a (ghost) fixed point $\mathcal{W}$, a session environment Δ, and a (ghost) list of environments $\mathcal{H}$ representing the *history* of the visited environments; the function returns an environment. Function `compliance` invokes `cstep` in order to accept or reject the environment Δ:

```
let[@ghost] rec cstep (o : oracle) (d : typEnv list)
  ((w : typEnvRedexes)[@ghost]) (delta: typEnv)
  ((history : typEnv list)[@ghost]): typEnv = ···
let[@ghost] compliance (o : oracle) (d : typEnv list)
  ((w : typEnvRedexes)[@ghost]) (delta : typEnv) :bool =
  try let m = cstep o d w delta [] in consumed m
  with | Fixpoint h → (* h = h0, e *) let e = last h in let h0 = pre h
  in mem_typEnv e h0 && sound e next | _ → raise NotCompliant
```

The behavioural specification of the functions is described in § 5.2. Ghost parameters are used both to provide a semantics to the fixed point mechanism and to prove the soundness of the accepted environments, and do not have computational interest: all ghost code referring to such parameters should be erased from the regular code after providing the proof effort [18,45].

In function `cstep` we use exceptions to tackle different behaviours of environments. In all cases but for exception `Fixpoint`, termination by raising an exception determines failure of establishing compliance.

Definition 13 (Positive exits). *Positive exits of function* `cstep` *are listed below. A positive exit implies that the parameter* Δ *of* `cstep` *satisfies compliance.*

name	param	exit	exception	positive
`cstep`	$\Omega, D, \mathcal{W}, \Delta, \mathcal{H}$	✓		✓
			`Fixpoint`	✓

W.r.t. the signatures of `cstep` and of closure in Definition 7, the non-ghost parameters are the same while the return type is different, because closure returns a set of environments. Remember that the aim of the returned set of environments is to establish *compliance* by verifying that all the final environments are not a communication mismatch, or a deadlock (cf. Definition 12). Function `cstep` achieves the same result by using exception handling and ghost parameters.

The body of `cstep` is recursive, and contains sub-calls of the form `cstep`$(\Omega, D\backslash_{\Delta}, \mathcal{W}, \Delta', (\mathcal{H}, \Delta))$: the first parameter Ω is the oracle and is the same in all calls; the second parameter $D\backslash_{\Delta}$ corresponds to the removal of Δ from the decreasing set D; the third parameter $\mathcal{W}$ is the fixed point and is the same in all calls; the fourth parameter Δ' is obtained by updating the type of one or of two participants returned by the oracle Ω (cf. Definition 5); the last parameter appends Δ to the history $\mathcal{H}$: in the remainder of the section, the *notation* $\mathcal{H}, \Delta$ indicates that Δ is the last environment visited in $\mathcal{H} \cup \{\Delta\}$.

5.2 Verification

The verification of function `compliance` relies on the behavioural specification and verification of function `cstep`, which in turn relies on auxiliary lemmas.

Figure 5 presents the behavioural specification of the implementation. The column **param** lists the input arguments of each function. The column **result** lists the result returned by each function. The column **variant** indicates the decreasing argument of `cstep`; note that `compliance` is not recursive. The column **requires** indicates the pre-conditions stated in terms of the parameters. The column **raises** indicates the formula holding for the argument carried by the exception; in the specification of `cstep` we omit exceptions asserting true. The column **ensures** indicates the post-condition stated in terms of the result.

There are two positive exits of function `cstep` establishing the *compliance* of the environment Δ received in input (cf. Definition 13): termination, and raising `Fixpoint`. The conditions holding when `cstep` raises an exception (cf. keyword **raises**) are discussed below while illustrating the verification process. Note that exceptions `Fixpoint`, `Deadlock`, `Wrongbranch`, `DecrNotFix` and `NotMinimal` carry the history $\mathcal{H}', \Delta'$, where Δ' is the last visited environment.

The predicate $\mathsf{sound}_{\Omega}(\Delta')$ relies on the result of the oracle and on Definition 9: if the oracle receives Δ' and returns one (cf. rule SD-REC) or two (cf. rule SD-COM) participants, and Δ' is not a mismatch, then Δ' is sound. The predicate $\mathsf{cons}(\Delta')$ says that all participants in the environment Δ' have type end.

name	**param**	**result**	**variant**	**requires**	**raises**	**ensures**
`cstep`	Ω	Δ_o	$\|D\|$	$\mathsf{fair}(\Omega)$	`OracleNotFair`$\Rightarrow$ false	$\mathsf{cons}(\Delta_o)$
	D			$\mathsf{isFix}(\mathcal{W}, \Delta, \|D\| * 2)$	`Fixpoint`$(\mathcal{H}', \Delta') \Rightarrow$ $\Delta' \in \mathcal{H}' \wedge \mathsf{sound}_{\Omega}(\Delta')$	
	$\mathcal{W}$			$D \cap \mathcal{H} = \emptyset$	`Deadlock`$(\mathcal{H}', \Delta') \Rightarrow$ $\Delta' \in \mathcal{H}' \wedge \mathsf{mismatch}(\Delta') \vee$ $\Omega(\Delta') = \mathsf{Ret}_0 \wedge \neg\mathsf{cons}(\Delta')$	
	Δ			$D \cup \mathcal{H} = \mathsf{comb}(\mathcal{W})$	`WrongBranch`$(\mathcal{H}', \Delta') \Rightarrow$ $\exists \mathtt{p}, \mathtt{q} \,.\, \Omega(\Delta') = \mathsf{Ret}_2(\mathtt{p}, \mathtt{q}) \wedge$ $\mathsf{mismatch}_2(\Delta'(\mathtt{p}), \Delta'(\mathtt{q}))$	
	$\mathcal{H}$			$\Delta \in \mathsf{comb}(\mathcal{W})$	`DecrNotFix`$(\mathcal{H}', \Delta') \Rightarrow$ $\Delta' \notin \mathcal{H}'$	
					`NotMinimal`$(\mathcal{H}', \Delta') \Rightarrow$ $\neg\mathsf{minimal}(\Delta')$	

$$\textsc{Sd-Rec}\,\frac{\Omega(\Delta) = \mathsf{Ret}_1(\mathtt{p}) \qquad \neg\mathsf{mismatch}(\Delta)}{\mathsf{sound}_{\Omega}(\Delta)} \qquad \textsc{Sd-Com}\,\frac{\Omega(\Delta) = \mathsf{Ret}_2(\mathtt{p}, \mathtt{q}) \qquad \neg\mathsf{mismatch}(\Delta)}{\mathsf{sound}_{\Omega}(\Delta)}$$

name	**param**	**result**	**requires**	**raises**	**ensures**
`compliance`	Ω	b	$\mathsf{fair}(\Omega)$	`NotCompliant`$\Rightarrow$ true	$b =$ true
	D		$\mathsf{isFix}(\mathcal{W}, \Delta, \|D\| * 2)$		
	$\mathcal{W}$		$D = \mathsf{comb}(\mathcal{W})$		
	Δ		$\Delta \in D$		

Fig. 5. Behavioural specification of the implementation

The post-condition (cf. keyword **ensures**) of `cstep` says that the returned environment is consumed: all participants have type end. For what concerns the pre-conditions of `cstep` (cf. keyword **requires**), the predicate $\mathsf{fair}(\Omega)$ implements Definition 5 by relying on constructors $\mathsf{Ret}_2, \mathsf{Ret}_1$ and Ret_0:

$$\begin{aligned}\mathsf{fair}(\Omega) \overset{\text{def}}{=} \forall \Delta \,.\, (\, & \forall \mathtt{p}\,\mathtt{q} \,.\, \Omega(\Delta) = \mathsf{Ret}_2(\mathtt{p}, \mathtt{q}) \Rightarrow \mathsf{top}(\Delta(\mathtt{p})) = \mathtt{q} \wedge \mathsf{top}(\Delta(\mathtt{q})) = \mathtt{p}) \wedge \\ & (\forall \mathtt{p} \,.\, \Omega(\Delta) = \mathsf{Ret}_1(\mathtt{p}) \Rightarrow \exists X\,T \,.\, \Delta(\mathtt{p}) = \mu X.T) \wedge \\ & (\forall X\,T\,\mathtt{r}\,\mathtt{p}\,\mathtt{q} \,.\, \Omega(\Delta) = \mathsf{Ret}_0 \Rightarrow \Delta(\mathtt{r}) \neq \mu X.T \wedge \\ & \neg(\mathsf{top}(\Delta(\mathtt{p})) = \mathtt{q} \wedge \mathsf{top}(\Delta(\mathtt{q}) = \mathtt{p})))\end{aligned}$$

The predicate $\mathsf{isFix}(\mathcal{W}, \Delta, n)$ says that $\mathcal{W}$ is a fixed point of Δ (up-to depth n). The core mechanism to analyse iso-recursive types and environments is to rely on fixed points $\mathcal{W}$ of type `typEnvRedexes`, that is a map from participants to all type redexes up-to depth n, and on the projection of all combinations of these mappings into a set of environments, denoted $\mathsf{comb}(\mathcal{W})$. The depth n indicates how many type transitions $T \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} T'$ are considered (cf. Figure 2); these include the unfolding of iso-recursive types $\mu X.T$ into $T\{\mu X.T/X\}$. Given a fixed point $\mathcal{W}$, we require that the decreasing set D and the history $\mathcal{H}$ partition the set $\mathsf{comb}(\mathcal{W})$ (cf. Figure 5, function `cstep`, keyword $\mathsf{requires}$, lines 3-4).

The pre-conditions of `compliance` mirror those of `cstep`, modulo the fact that there is no history. The post-condition of `compliance` ensures that the function returns true by exploiting (1) the post-condition of `cstep` and (2) the formula holding when `cstep` raises `Fixpoint`. The exceptional exit of `compliance` occurs when raising `NonCompliant`, thus rejecting the input environment Δ.

Termination. The first result establishes that function `cstep` terminates. We instruct [45] to use $|D|$ as decreasing measure, cf. the keyword $\mathsf{variant}$ in the function specification of Figure 5, and obtain the desired result automatically.

Absence of communication mismatches. In order to show that environments accepted by `compliance` are mismatch-free, we ensure that positive exits of function `cstep` (cf. Definition 13) carry environments that are not communication mismatches (cf. Definition 9) by inspecting `cstep`'s contract in Figure 5.

The first positive exit is termination: `cstep` returns Δ_o. The contract's clause with keyword $\mathsf{ensures}$ establishes that Δ_o is consumed: by definition, Δ_o is not a mismatch. The second positive exit corresponds to the exception `Fixpoint`: the exceptions carries the history $\mathcal{H}', \Delta'$, where Δ' is the last visited environment. The clause raises establishes that $\Delta' \in \mathcal{H}'$ and that Δ' is sound. By inversion of rules Sd-Rec, Sd-Com, we obtain that Δ' is not a mismatch. □

The automated verification is performed in [45] and relies on the predicate $\mathsf{mismatch}_2(T_1, T_2)$ (cf. Figure 5) to deal with wrong choices of sums: intuitively, the predicate follow Definition 9 by using types rather than participants.

Absence of deadlocks. Similarly, we show that positive exits of `cstep` of Definition 13 correspond to absence of deadlocks of Definition 10.

The first positive exit occurs when function `cstep` returns Δ_o. The clause with keyword $\mathsf{ensures}$ establishes that Δ_o is consumed: by definition, Δ_o is not a deadlock. The second positive exit is raising exception `Fixpoint`; the exception carries the history $\mathcal{H}', \Delta'$, where Δ' is the last visited environment. From the contract's clause with keyword raises, we infer that $\Delta' \in \mathcal{H}'$ and that Δ' is sound.

By inversion of rules Sd-Rec, Sd-Com, we obtain that two cases arise: (1) there is a participant $\mathtt{p}$ s.t. $\Omega(\Delta') = \mathsf{Ret}_1(\mathtt{p})$ and Δ' is not a mismatch; (2) there are participants $\mathtt{p}$, $\mathtt{q}$ such that $\Omega(\Delta') = \mathsf{Ret}_2(\mathtt{p}, \mathtt{q})$ and Δ' is not a mismatch. We show that in both cases (1) and (2) we have that Δ' can do a transition.

1. By the fairness pre-condition of `cstep`, we obtain
$$\Omega(\Delta') = \mathsf{Ret}_1(\mathtt{p}) \Rightarrow \exists X\, T\,.\, \Delta'(\mathtt{p}) = \mu X.T$$
We apply E-Rec, Se-Rec of Figure 2 and find Δ'' s.t. $\Delta' \xrightarrow{\tau} \Delta''$.
2. By the fairness pre-condition of `cstep`, we obtain
$$\Omega(\Delta') = \mathsf{Ret}_2(\mathtt{p}, \mathtt{q}) \Rightarrow \mathsf{top}(T_{\mathtt{p}}) = \mathtt{q} \wedge \mathsf{top}(T_{\mathtt{q}}) = \mathtt{p}$$
where $\Delta'(\mathtt{p}) \overset{\text{def}}{=} T_{\mathtt{p}}$ and $\Delta'(\mathtt{q}) \overset{\text{def}}{=} T_{\mathtt{q}}$.
By inversion of $\mathsf{top}(T_{\mathtt{p}})$ and $\mathsf{top}(T_{\mathtt{q}})$ (cf. Definition 5), we obtain that $T_{\mathtt{p}} = \&_{i \in I} \mathtt{q}?l_i(S_i).T_i$ or $T_{\mathtt{p}} = \oplus_{i \in I} \mathtt{q}!l_i(S_i).T_i$, and $T_{\mathtt{q}} = \&_{j \in J} \mathtt{p}?l_j(S_j).T_j$ or $T_{\mathtt{q}} = \oplus_{j \in J} \mathtt{p}!l_j(S_j).T_j$.
By hypothesis, Δ' is not a mismatch: Definition 9 ensures that two sub-cases arise: $T_{\mathtt{p}} = \&_{i \in I} \mathtt{q}?l_i(S_i).T_i$ and $T_{\mathtt{q}} = \oplus_{j \in J} \mathtt{p}!l_j(S_j).T_j$ and $\mathcal{L}(T_{\mathtt{p}}) \cap \mathcal{L}(T_{\mathtt{q}}) \neq \emptyset$, or $T_{\mathtt{p}} = \oplus_{i \in I} \mathtt{q}!l_i(S_i).T_i$, and $T_{\mathtt{q}} = \&_{j \in J} \mathtt{p}?l_j(S_j).T_j$ and $\mathcal{L}(T_{\mathtt{p}}) \cap \mathcal{L}(T_{\mathtt{q}}) \neq \emptyset$. In both cases we apply Se-Com of Figure 2 and find α, Δ'' s.t. $\Delta' \xrightarrow{\alpha} \Delta''$. □

6 Related Work

To the best of our knowledge, only few works follow an iso-recursive approach to session types. [31] proposes a decentralized analysis of multiparty protocols that is based on a typed asynchronous π-calculus relying on the notion of router processes; deadlock-freedom is established by following the priority-based approach of session types [13]. The rule to type check recursion types the continuation by unfolding iso-recursive types and lifting priorities to a common greater highest priority. Finally, type preservation holds up to unfolding (cf. [31, Theorem 2]).

[36] studies iso-recursive and equi-recursive subtyping for binary sessions. Session types are interpreted as propositions of multiplicative/additive linear logic extended with least and greatest fixed points (cf. [8,54]). The typing rules correspond to the proof rules in [3], and include the unfolding of least and greatest fixed points. The authors compare the two subtyping relations, and note that the relations preserve not only the usual safety properties, but also termination.

Many recent papers [56,57,58,59,48,43,39] rely on iso-recursive types for variants of the λ-calculus, following the seminal work on Amber rules [1]. While the setting is different from ours, these papers provide several insights on the advantage of iso-recursive types and on their algorithmic implementation and mechanised verification. Previous papers [6,2] studied iso-recursive types for a concurrent λ-calculus that can be seen as the foundational theory of core $F^\sharp$.

As mentioned above, iso-recursive types have been first studied formally in the setting of Amber rules [1]. Pierce's book [47] further discusses the differences between iso-recursive and equi-recursive types.

Future Work. Our plans go along two directions: completing the study in the paper and extending the language model and the type analysis.

Towards completion, we plan to conclude the mechanisation of subject reduction in Coq, and to compare the performance of compliance checking in OCaml with the verification of deadlock freedom in bottom-up approaches (cf. [52]) relying on model-checking [49], eventually considering a realistic testing suite involving multiple participants and interactions (cf. [49, Table 2]).

For what concerns extensions, there are two main features we are interested in: session delegation and asynchronous subtyping for multiparty session types.

Handling session delegation in session types is challenging and might require type constructors [25] or session channel decorations [22,15,53] to preserve type soundness. Our plan is to enforce soundness at the type level, without affecting the programmer's syntax.

Asynchronous subtyping (e.g. [24]) is known to be undecidable for more than two participants. We envision to overcome this obstacle to an algorithmic solution by considering a maximal depth of the search of the asynchronous outputs that can be anticipated, similarly to the bound on recursion in [12].

Acknowledgements. We thank the reviewers for detailed and helpful comments. This work is partially supported by EPSRC EP/T006544/2, EP/N027833/2, EP/T014709/2, EP/Y005244/1, EP/V000462/1, Horizon EU TaRDIS 101093006, Advanced Research and Invention Agency (ARIA), and a grant from the Simons Foundation.

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Monographs in Computer Science, Springer (1996). https://doi.org/10.1007/978-1-4419-8598-9
2. Backes, M., Hritcu, C., Maffei, M.: Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations. J. Comput. Secur. **22**(2), 301–353 (2014). https://doi.org/10.3233/JCS-130493
3. Baelde, D., Doumane, A., Kuperberg, D., Saurin, A.: Bouncing threads for circular and non-wellfounded proofs: Towards compositionality with circular proofs. In: Baier, C., Fisman, D. (eds.) LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022. pp. 63:1–63:13. ACM (2022). https://doi.org/10.1145/3531130.3533375
4. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14
5. Barwell, A.D., Scalas, A., Yoshida, N., Zhou, F.: Generalised multiparty session types with crash-stop failures. In: Klin, B., Lasota, S., Muscholl, A. (eds.) 33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland. LIPIcs, vol. 243, pp. 35:1–35:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPICS.CONCUR.2022.35
6. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. ACM Trans. Program. Lang. Syst. **33**(2), 8:1–8:45 (2011). https://doi.org/10.1145/1890028.1890031
7. Brun, M.A.L., Dardha, O.: Magπ: Types for failure-prone communication. In: Wies, T. (ed.) Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13990, pp. 363–391. Springer (2023). https://doi.org/10.1007/978-3-031-30044-8_14
8. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6269, pp. 222–236. Springer (2010). https://doi.org/10.1007/978-3-642-15375-4_16
9. Charguéraud, A., Filliâtre, J., Lourenço, C., Pereira, M.: GOSPEL - providing OCaml with a formal specification language. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11800, pp. 484–501. Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_29
10. ContainerSSH: DevLog: SSH authentication via OAuth2, https://containerssh.io/v0.5/blog/2021/04/13/devlog-oauth2/
11. Coq development team: Reference manual, https://coq.inria.fr/doc/V8.20.0/refman/
12. Cutner, Z., Yoshida, N., Vassor, M.: Deadlock-free asynchronous message reordering in rust with multiparty session types. In: Lee, J., Agrawal, K., Spear, M.F. (eds.) PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of

Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022. pp. 246–261. ACM (2022). https://doi.org/10.1145/3503221.3508404
13. Dardha, O., Gay, S.J.: A new linear logic for deadlock-free session-typed processes. In: Baier, C., Lago, U.D. (eds.) Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10803, pp. 91–109. Springer (2018). https://doi.org/10.1007/978-3-319-89366-2_5
14. Demangeon, R., Honda, K.: Full abstraction in a subtyped pi-calculus with linear types. In: Katoen, J., König, B. (eds.) CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6901, pp. 280–296. Springer (2011). https://doi.org/10.1007/978-3-642-23217-6_19
15. Dezani-Ciancaglini, M., Drossopoulou, S., Mostrous, D., Yoshida, N.: Objects and session types. Inf. Comput. **207**(5), 595–641 (2009). https://doi.org/10.1016/J.IC.2008.03.028
16. Dross, C., Conchon, S., Kanig, J., Paskevich, A.: Adding decision procedures to SMT solvers using axioms with triggers. J. Autom. Reason. **56**(4), 387–457 (2016). https://doi.org/10.1007/S10817-015-9352-2
17. Ekici, B., Yoshida, N.: Completeness of asynchronous session tree subtyping in coq. In: Bertot, Y., Kutsia, T., Norrish, M. (eds.) 15th International Conference on Interactive Theorem Proving, ITP 2024, September 9-14, 2024, Tbilisi, Georgia. LIPIcs, vol. 309, pp. 13:1–13:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024). https://doi.org/10.4230/LIPICS.ITP.2024.13
18. Filliâtre, J., Gondelman, L., Paskevich, A.: The spirit of ghost code. Formal Methods Syst. Des. **48**(3), 152–174 (2016). https://doi.org/10.1007/S10703-016-0243-X
19. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
20. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3), 17 (2007). https://doi.org/10.1145/1232420.1232424
21. Gay, S., Ravara, A. (eds.): Behavioural Types: from Theory to Tools. River Publishers (2017), https://doi.org/10.13052/rp-9788793519817
22. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta Informatica **42**(2-3), 191–225 (2005). https://doi.org/10.1007/S00236-005-0177-Z
23. Ghilezan, S., Jaksic, S., Pantovic, J., Scalas, A., Yoshida, N.: Precise subtyping for synchronous multiparty sessions. J. Log. Algebraic Methods Program. **104**, 127–173 (2019). https://doi.org/10.1016/J.JLAMP.2018.12.002
24. Ghilezan, S., Pantović, J., Prokić, I., Scalas, A., Yoshida, N.: Precise subtyping for asynchronous multiparty sessions. ACM Trans. Comput. Logic **24**(2) (nov 2023). https://doi.org/10.1145/3568422
25. Giunti, M., Vasconcelos, V.T.: A linear account of session types in the pi calculus. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory, 21th

International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6269, pp. 432–446. Springer (2010). https://doi.org/10.1007/978-3-642-15375-4_30

26. Giunti, M., Vasconcelos, V.T.: Linearity, session types and the pi calculus. Math. Struct. Comput. Sci. **26**(2), 206–237 (2016). https://doi.org/10.1017/S0960129514000176
27. Giunti, M., Yoshida, N.: Iso-Recursive Multiparty Sessions and their Automated Verification – Technical Report (2025). https://doi.org/10.48550/ARXIV.2501.17778
28. van Glabbeek, R., Höfner, P., Horne, R.: 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. pp. 1–13. IEEE (2021). https://doi.org/10.1109/LICS52264.2021.9470531
29. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press (2014), https://mitpress.mit.edu/books/modeling-and-analysis-communicating-systems
30. Harvey, P., Fowler, S., Dardha, O., Gay, S.J.: Multiparty session types for safe runtime adaptation in an actor language. In: Møller, A., Sridharan, M. (eds.) 35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference). LIPIcs, vol. 194, pp. 10:1–10:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ECOOP.2021.10
31. van den Heuvel, B., Pérez, J.A.: A decentralized analysis of multiparty protocols. Sci. Comput. Program. **222**, 102840 (2022). https://doi.org/10.1016/J.SCICO.2022.102840
32. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings. Lecture Notes in Computer Science, vol. 715, pp. 509–523. Springer (1993). https://doi.org/10.1007/3-540-57208-2_35
33. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1381, pp. 122–138. Springer (1998). https://doi.org/10.1007/BFB0053567
34. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 273–284. ACM (2008). https://doi.org/10.1145/1328438.1328472
35. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016). https://doi.org/10.1145/2827695
36. Horne, R., Padovani, L.: A logical account of subtyping for session types. J. Log. Algebraic Methods Program. **141**, 100986 (2024). https://doi.org/10.1016/J.JLAMP.2024.100986
37. Imai, K., Neykova, R., Yoshida, N., Yuen, S.: Multiparty session programming with global protocol combinators. In: Hirschfeld, R., Pape, T. (eds.) 34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference). LIPIcs, vol. 166, pp. 9:1–9:30.

Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPICS.ECOOP.2020.9

38. Lange, J., Ng, N., Toninho, B., Yoshida, N.: A static verification framework for message passing in go using behavioural types. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (eds.) Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. pp. 1137–1148. ACM (2018). https://doi.org/10.1145/3180155.3180157
39. Ligatti, J., Blackburn, J., Nachtigal, M.: On subtyping-relation completeness, with an application to iso-recursive types. ACM Trans. Program. Lang. Syst. **39**(1), 4:1–4:36 (2017). https://doi.org/10.1145/2994596
40. Meyer, B.: Applying "design by contract". Computer **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
41. Milner, R.: Communication and concurrency. PHI Series in computer science, Prentice Hall (1989)
42. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
43. Patrignani, M., Martin, E.M., Devriese, D.: On the semantic expressiveness of recursive types. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021). https://doi.org/10.1145/3434302
44. Pereira, M.: Practical Deductive Verification of OCaml Programs. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part II. Lecture Notes in Computer Science, vol. 14934, pp. 518–542. Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_29
45. Pereira, M., Ravara, A.: Cameleer: A deductive verification tool for OCaml. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 677–689. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_31
46. Peters, K., Yoshida, N.: Separation and encodability in mixed choice multiparty sessions. In: Sobocinski, P., Lago, U.D., Esparza, J. (eds.) Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2024, Tallinn, Estonia, July 8-11, 2024. pp. 62:1–62:15. ACM (2024). https://doi.org/10.1145/3661814.3662085
47. Pierce, B.C.: Types and programming languages. MIT Press (2002)
48. Rossberg, A.: Mutually iso-recursive subtyping. Proc. ACM Program. Lang. **7**(OOPSLA2), 347–373 (2023). https://doi.org/10.1145/3622809
49. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. Proc. ACM Program. Lang. **3**(POPL), 30:1–30:29 (2019). https://doi.org/10.1145/3290343
50. Scalas, A., Yoshida, N., Benussi, E.: Verifying message-passing programs with dependent behavioural types. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 502–516. ACM (2019). https://doi.org/10.1145/3314221.3322484

51. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Halatsis, C., Maritsas, D.G., Philokyprou, G., Theodoridis, S. (eds.) PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings. Lecture Notes in Computer Science, vol. 817, pp. 398–413. Springer (1994). https://doi.org/10.1007/3-540-58184-7_118
52. Udomsrirungruang, T., Yoshida, N.: Top-down or bottom-up? Complexity analyses of synchronous multiparty session types. Proc. ACM Program. Lang. **9**(POPL) (Jan 2025). https://doi.org/10.1145/3704872
53. Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. **217**, 52–70 (2012). https://doi.org/10.1016/J.IC.2012.05.002
54. Wadler, P.: Propositions as sessions. J. Funct. Program. **24**(2-3), 384–418 (2014). https://doi.org/10.1017/S095679681400001X
55. Yoshida, N.: Programming language implementations with multiparty session types. In: de Boer, F.S., Damiani, F., Hähnle, R., Johnsen, E.B., Kamburjan, E. (eds.) Active Object Languages: Current Research Trends, Lecture Notes in Computer Science, vol. 14360, pp. 147–165. Springer (2024). https://doi.org/10.1007/978-3-031-51060-1_6
56. Zhou, L., Oliveira, B.C.d.S.: Quicksub: Efficient iso-recursive subtyping. Proc. ACM Program. Lang. **9**(POPL) (Jan 2025). https://doi.org/10.1145/3704869
57. Zhou, L., Wan, Q., d. S. Oliveira, B.C.: Full iso-recursive types. Proc. ACM Program. Lang. **8**(OOPSLA2), 192–221 (2024). https://doi.org/10.1145/3689718
58. Zhou, L., Zhou, Y., d. S. Oliveira, B.C.: Recursive subtyping for all. Proc. ACM Program. Lang. **7**(POPL), 1396–1425 (2023). https://doi.org/10.1145/3571241
59. Zhou, Y., Zhao, J., d. S. Oliveira, B.C.: Revisiting iso-recursive subtyping. ACM Trans. Program. Lang. Syst. **44**(4), 24:1–24:54 (2022). https://doi.org/10.1145/3549537