Type Congruence, Duality and Iso-Recursive Binary Session Types*

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

University of Oxford, UK

Abstract. Session types express a *protocol* which specifies the order and type of messages exchanged by concurrently executing processes. Prior work has been adapted to many frameworks including Agha's work [10], which has successfully extended session types to model distributed asynchronous multi-actors. Most works on session types, including Agha's work, take an equi-recursive approach and do not distinguish among a recursive type and its unfolding. One main problem of an equi-recursive type system is that its mechanisation in proof assistants is utterly complex and eventually requires co-induction. To overcome this problem, this paper presents an *iso-recursive* type system for *binary* sessions based on a novel *congruence on types* which relates recursive types and their unfolding. Our system based on type congruence enables to use a simple syntactic-directed *duality* without complicating the typability of processes. We mechanise the type congruence relation in Rocq without resorting to coinductive types, and use the proof assistant to show that our iso-recursive typing system satisfies subject reduction.

1 Introduction

Session types [28,50,30] are an effective framework to govern the behaviour of distributed processes that interact with message-passing. Their process algebraic properties based on the calculi of mobile processes (the π-calculus) [39,40] have been studied and used for about 30 years. As Robin Milner stated [38], mobile processes act as a *canonical encapsulation* of the concurrent object paradigm [53] represented by *actors* by Agha and Hewitt [2]. Actors are independent agents (processes) which perform computations by purely exchanging asynchronous messages. The actors' model plays a key role for developing the syntactically "minimum", expressive and implementable π-calculus (the asynchronous π-calculus) [29] and its semantic theories [31], for measuring the expressiveness of a family of process algebras, and for designing distributed implementations [5]. To cope with asynchronous actor semantics, Agha, with Charalambides and Dinges, has extended multiparty session types for specifying parameterised, multi-actor protocols with inherent asynchrony [10]. This powerful

* This work is partially supported by EPSRC EP/T006544/2, EP/N027833/2, EP/T014709/2, EP/Y005244/1, EP/V000462/1, EP/X015955/1, Horizon EU TaRDIS 101093006 and Advanced Research and Invention Agency (ARIA) Safeguarded AI.

framework enables us to express a large class of asynchronous communication protocols, including the sliding window protocol, and is successfully integrated into an actor programming language.

This paper attempts to propose a tractable but expressive typing system for session processes which are based on a simple type duality relation. In most works on session types, including Agha's work, recursive types follow an *equi-recursive* view [44] and represent infinite trees that are manipulated *co-inductively*. This representation does not have a direct counterpart in eager programming languages, as modelled by an object calculus [1], which typically resort to *iso-recursive* types [44] 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, session types are embedded in non-lazy languages by encoding equi-recursive types; for instance, [33] defines infinite sequence of types as polymorphic lenses [16] by using OCaml generalised algebraic data types.

Our proposal to overcome this problem consists in introducing a theory of *iso-recursive session types* relying on a type system that uses a novel notion of *type congruence* to relate the types of dual sessions. This contribution complements recent results [25] presenting a theory of iso-recursive multiparty session types. The theory in [25] relies on the bottom-up approach known as *generalised multiparty session types*, e.g. [34,49,48,6,19,43,27,8], and decides deadlock-freedom without using global types. Differently from previous work, it considers iso-recursive types and computes the properties of session environments in the type system, instead of assessing these properties with model checkers (cf. [48]).

In this paper, we account for type checking the parallel composition of sessions typed with *folded* and *unfolded* dual iso-recursive session types by introducing a *type congruence* that is closed under simple *syntax-directed session type duality*, and under *labelled transitions* of types. The following example illustrates these ideas.

Equi-recursive vs iso-recursive types: OAuth2 example. Consider a simplified version of the OAuth 2.0 protocol [48] relying on *two dual participants*. 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 and unit values. The protocol says that the client ($\mathtt{c}$) sends the password ($\mathtt{pwd}$, carrying a string) to the authorisation server ($\mathtt{a}$) and the session restarts, or $\mathtt{c}$ sends $\mathtt{quit}$ to $\mathtt{a}$, and the session ends.

$$T_{\mathtt{c}} \stackrel{\text{def}}{=} \mu X.(\mathtt{a}!\mathsf{pwd}(\mathsf{str}).X + \mathtt{a}!\mathsf{quit}(\mathsf{unit}).\mathsf{end})$$

$$T_{\mathtt{a}} \stackrel{\text{def}}{=} \mu X.R_{\mathtt{a}} \qquad R_{\mathtt{a}} \stackrel{\text{def}}{=} \mathtt{c}?\mathsf{pwd}(\mathsf{str}).X + \mathtt{c}?\mathsf{quit}(\mathsf{unit}).\mathsf{end}$$

Most type disciplines for binary sessions rely on *equi-recursion* and *type duality*, e.g. [30,17,23]. Equi-recursion establishes that a recursive type and its unfolding are *equal*. Consider the types $T_{\mathsf{a}}^* \stackrel{\text{def}}{=} R_{\mathsf{a}}\{T_{\mathsf{a}}/X\}$, $T_{\mathsf{a}}^{**} \stackrel{\text{def}}{=} R_{\mathsf{a}}\{T_{\mathsf{a}}^*/X\}$, and so on. Following the equi-recursive approach, we have $T_{\mathsf{a}} = T_{\mathsf{a}}^* = T_{\mathsf{a}}^{**} = \cdots$. We note that mechanising equi-recursive session types in proof assistants is quite complex, and eventually relies on co-induction [14]. Duality swaps participants, and ?/! with !/?, in a type T, noted $\overline{T}$: it verifies the compatibility of the two participants of a session. The composition of two end-points of the communication is allowed only if their types are dual: this is mandatory for subject reduction [22,51,24].

To show an example of *equi-recursive session typing*, consider the processes implementing the client (P_{c}) and the server (P_{a}), which mirror their types T_{c} and T_{a}, respectively, but that the payload of input and output are (bound) variables and expressions, respectively, and that there is a process variable χ:

$$P_{\mathsf{c}} \stackrel{\text{def}}{=} \mu\chi.(\mathsf{a}!\mathsf{pwd}\langle\text{“fido”}\rangle.\chi + \mathsf{a}!\mathsf{quit}\langle\rangle) \qquad P_{\mathsf{a}} \stackrel{\text{def}}{=} \mu\chi.Q_{\mathsf{a}} \qquad Q_{\mathsf{a}} \stackrel{\text{def}}{=} \mathsf{c}?\mathsf{pwd}(x).\chi + \mathsf{c}?\mathsf{quit}(x)$$

The composition of processes P_{c} and $P_{\mathsf{a}}^* \stackrel{\text{def}}{=} Q_{\mathsf{a}}\{P_{\mathsf{a}}/\chi\}$ is typed by the rule for parallel composition, where the type $\bot$ indicates that no further interactions are possible [30]. The equality $\overline{T_{\mathsf{c}}} = T_{\mathsf{a}}^*$ is interpreted on regular infinite trees [12]: “to check whether a given end point type T_1 is dual of another type T_2, we first build the dual of T_1 and then check that the thus obtained type is equivalent to T_2” [24, p. 213].

$$\textsc{Equi-T-Sess}\ \frac{\Gamma \vdash P_{\mathsf{c}} : T_{\mathsf{c}} \qquad \Gamma \vdash P_{\mathsf{a}}^* : T_{\mathsf{a}}^* \qquad \overline{T_{\mathsf{c}}} = T_{\mathsf{a}}^*}{\Gamma \Vdash \mathsf{c} \lhd P_{\mathsf{c}} \parallel \mathsf{a} \lhd P_{\mathsf{a}}^* : \bot}$$

The approach proposed in this paper relies on *iso-recursive session types* and is mechanised in Rocq without resorting to coinductive types (cf. command `CoInductive` [46]). First, we define a *congruence* on types relating folded and unfolded types, noted $\equiv$. Then, to type check the parallel composition of the client $\mathsf{c} \lhd P_{\mathsf{c}}$ and of the server $\mathsf{a} \lhd P_{\mathsf{a}}^*$, we *provide a proof that* $\overline{T_{\mathsf{c}}} \equiv T_{\mathsf{a}}^*$.

$$\textsc{Iso-T-Sess}\ \frac{\Gamma \vdash P_{\mathsf{c}} : T_{\mathsf{c}} \qquad \Gamma \vdash P_{\mathsf{a}}^* : T_{\mathsf{a}}^* \qquad \overline{T_{\mathsf{c}}} \equiv T_{\mathsf{a}}^*}{\Gamma \Vdash \mathsf{c} \lhd P_{\mathsf{c}} \parallel \mathsf{a} \lhd P_{\mathsf{a}}^* : \bot}$$

We show that the typing system based on rule Iso-T-Sess is sound by relying on the “types as processes” approach (cf. [37]) and by introducing labelled transition semantics of types.

Outline § 2 introduces the syntax and the labelled transition semantics of iso-recursive session processes. § 3 presents a type congruence relating folded and unfolded types. The iso-recursive type system is developed in § 4, along with the type duality involution. § 5 is devoted to the proof of subject reduction. We conclude in § 6 by briefly discussing related work.

All the *results* of the paper are *mechanised in Rocq* and are publicly available in an *online repository* [21]. Along the paper, we map definitions and lemmas to their Rocq counterpart by means of *hyperlinks* denoted by the *Roc bird symbol* .

2 Iso-Recursive Sessions

In this section, we present the syntax and the semantics of binary 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). 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 subterm for any $n \geq 1$ [44, p. 300], which can be alternatively stated as type variables must occur guarded [13].

Definition 1 (Syntax of types and processes).

$S := \mathsf{nat} \mid \mathsf{int} \mid \mathsf{str} \mid \mathsf{bool} \mid \mathsf{unit}$	Sorts
$T := \mathtt{r}!l(S).T \mid \mathtt{r}?l(S).T \mid T + T \mid \mathsf{end} \mid \mu X.T \mid X \mid \bot$	Types
$v := n \mid z \mid s \mid () \mid \mathsf{true} \mid \mathsf{false}$	Values
$e := v \mid e_1 \wedge e_2 \mid e_1 \vee e_2 \mid \neg e \mid e_1 == e_2 \mid e_1 < e_2$	Expressions
$P := \mathtt{r}!l\langle e\rangle.P \mid \mathtt{r}?l(x).P \mid P + P \mid \mu\chi.P \mid \chi \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid \mathbf{0}$	Processes
$\mathcal{M} := \mathtt{r} \lhd P \mid (\mathtt{p} \lhd P \parallel \overline{\mathtt{p}} \lhd Q)$	Sessions

We use $\mathtt{p}, \mathtt{q}, \mathtt{r}$ to range over the set $\mathcal{P}$ of *participants*, and let *dual*: $\mathcal{P} \to \mathcal{P}$ be an involution, noted $\overline{\mathtt{r}}$. We use n to range over *natural values*, z to range over *integer values*, and s to range over *string values*; () indicates the *empty value*. We use l to range over *labels*, i, j to range over indexes (natural numbers), X to range over *type variables*, x to range over *variables*, and χ to range over *process variables*.

A *thread* is the decoration of a process P with a participant $\mathtt{p}$, noted $\mathtt{p} \lhd P$. *Sessions* $\mathcal{M}$ belong to the set $\mathbb{M}$; types T belong to the set $\mathcal{T}$. To ease the presentation, $\mathcal{M}$ contains at most two threads having dual participants: having multiple pairs of threads with dual participants does not affect our results. Type $\bot$ is reserved for closing the composition of two threads (cf. § 4).

The constructor μ is a *binder* in types and processes: 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 $\mathtt{r}?l(x).P$. *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 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 substitution of free occurrences of variable x in process P with a value v, written $P\{v/x\}$.

The symbol $=$ is reserved for Leibniz equality.

Definition 2 (Session notation).

$$\oplus_{i\in I}\mathtt{r}!l_i(S_i).T_i \stackrel{def}{=} \mathtt{r}!l_1(S_1).T_1 + \cdots + \mathtt{r}!l_n(S_n).T_n \quad I = (1,\ldots,n), n \geq 1$$

$$\&_{i\in I}\mathtt{r}?l_i(S_i).T_i \stackrel{def}{=} \mathtt{r}?l_1(S_1).T_1 + \cdots + \mathtt{r}?l_n(S_n).T_n \quad I = (1,\ldots,n), n \geq 1$$

$$\text{WF-Inp}\ \frac{\mathsf{wf}(T)}{\mathsf{wf}(\mathtt{r}?l(S).T)} \qquad \text{WF-Out}\ \frac{\mathsf{wf}(T)}{\mathsf{wf}(\mathtt{r}!l(S).T)}$$

$$\text{WF-Sum}\ \frac{\mathsf{nodup}(\mathsf{labels}(T_1+T_2)) \quad \mathsf{polarity}(T_1+T_2)\downarrow \quad \mathsf{wf}(T_1) \quad \mathsf{wf}(T_2)}{\mathsf{wf}(T_1+T_2)}$$

$$\text{WF-End}\ \frac{}{\mathsf{wf}(\mathsf{end})} \qquad \text{WF-Rec}\ \frac{\mathsf{wf}(T)}{\mathsf{wf}(\mu X.T)} \qquad \text{WF-Var}\ \frac{}{\mathsf{wf}(X)}$$

Fig. 1. Well-formed types

2.1 Well-Formed Types

We only consider *well-formed* types implementing the session notation in Definition 2. Well-formedness is decided by collecting the labels of types in multi-sets; the inference system $\mathsf{wf}(T)$ is in Figure 1.

Intuitively, a sum type T_1+T_2 is *well-formed* when it has no duplicated labels, T_1 and T_2 have the same polarity, and T_i does not contain type $\bot$, $i \in \{1,2\}$.

Let $\{|l_1,\ldots,l_n|\}$ denote *multisets* of labels, and let $\mathsf{nodup}(\{|l_1,\ldots,l_n|\})$ hold whenever l_i is unique, for all $i \in \{1,\ldots,n\}$.

Let labels be a function mapping types to multisets of labels, and assume $\mathsf{labels}(\mathtt{r}?l(S).T) \stackrel{\text{def}}{=} \{|l|\}$, $\mathsf{labels}(\mathtt{r}!l(S).T) \stackrel{\text{def}}{=} \{|l|\}$, and $\mathsf{labels}(T_1+T_2) \stackrel{\text{def}}{=} \mathsf{labels}(T_1) \uplus \mathsf{labels}(T_2)$, and $\mathsf{labels}(T) \stackrel{\text{def}}{=} \emptyset$ otherwise.

Let $\mathsf{polarity}$ be a partial function mapping types to polarities $p \in \{!,?\}$, and let $\mathsf{polarity}(T)\downarrow$ whenever $\mathsf{polarity}$ is defined on type T. Let $\mathsf{polarity}(\mathtt{r}?l(S).T) \stackrel{\text{def}}{=} ?$, $\mathsf{polarity}(\mathtt{r}!l(S).T) \stackrel{\text{def}}{=} !$, and $\mathsf{polarity}(T_1+T_2) \stackrel{\text{def}}{=} p$ whenever $\mathsf{polarity}(T_1)=p$ and $\mathsf{polarity}(T_2)=p$, and $\mathsf{polarity}(T)$ be undefined otherwise.

Well-formed types contain sums having the same polarity and without duplicated labels, and do not contain the type $\bot$, which is reserved for the typing system (cf. § 4). The formal definition is in Figure 1.

Example 1. Examples of well-formed types include $\mathtt{p}?l(\mathsf{nat}).\mathtt{p}!l(\mathsf{bool}).T_1$, and $\mathtt{p}?l_1(\mathsf{str}).T_1 + \mathtt{p}?l_2(\mathsf{int}).T_2 + \mathtt{p}?l_3(\mathsf{unit}).T_3$, and $\mathtt{p}!l_1(\mathsf{str}).T_1 + \mathtt{p}!l_2(\mathsf{int}).T_2 + \mathtt{p}!l_3(\mathsf{unit}).T_3$, and $\mu X.(\mathtt{p}?l_1(\mathsf{str}).X + \mathtt{p}?l_2(\mathsf{int}).T_2)$, whenever l_1, l_2 and l_3 are distinct, and T_1, T_2 and T_3 are well-formed.

A first example of ill-formed type is $\mathtt{p}?l(\mathsf{str}).T_1 + \mathtt{p}?l(\mathsf{int}).T_2$, because the label l is used twice.

The type $T \stackrel{\text{def}}{=} \mu X.(\mathtt{p}?l_1(\mathsf{str}).X) + \mathtt{p}?l_2(\mathsf{int}).T_1$ is ill-formed, because $\mathsf{polarity}(T)$ is undefined: this follows from $\mathsf{polarity}(\mu X.(\mathtt{p}?l_1(\mathsf{str}).X))$ undefined. To recover well-formedness, whenever $\mathtt{p}?l_2(\mathsf{int}).T_1$ is well-formed and does not contain free occurrences of X (which can be enforced by alpha-renaming $\mu X.(\mathtt{p}?l_1(\mathsf{str}).X)$), we can lift recursion up: $\mathsf{wf}(\mu X.(\mathtt{p}?l_1(\mathsf{str}).X + \mathtt{p}?l_2(\mathsf{int}).T_1))$.

The type $\mathtt{p}!l(\mathsf{int}).(\mathtt{p}?l(\mathsf{bool}).\mathsf{end}+\mathsf{end})$ is ill-formed, because $\mathsf{polarity}(\mathtt{p}?l(\mathsf{bool}).\mathsf{end}+\mathsf{end})$ is undefined: this follows from $\mathsf{polarity}(\mathsf{end})$ undefined.

Transition rules for sessions: $\boxed{\mathcal{M} \xrightarrow{\alpha} \mathcal{M}}$

$$\text{R-Inp}\ \frac{}{\mathtt{p} \lhd \overline{\mathtt{p}}?l(x).P \xrightarrow{\overline{\mathtt{p}}?l(v)} \mathtt{p} \lhd P\{v/x\}} \qquad \text{R-Out}\ \frac{e \downarrow v}{\mathtt{p} \lhd \overline{\mathtt{p}}!l\langle e\rangle.P \xrightarrow{\overline{\mathtt{p}}!l\langle v\rangle} \mathtt{p} \lhd P}$$

$$\text{R-Sum-L}\ \frac{\mathtt{r} \lhd P \xrightarrow{\alpha} \mathtt{r} \lhd P'}{\mathtt{r} \lhd P + Q \xrightarrow{\alpha} \mathtt{r} \lhd P'}$$

$$\text{R-Com}\ \frac{\mathtt{p} \lhd P \xrightarrow{\overline{\mathtt{p}}?l(v)} \mathtt{p} \lhd P' \qquad \overline{\mathtt{p}} \lhd Q \xrightarrow{\mathtt{p}!l\langle v\rangle} \overline{\mathtt{p}} \lhd Q'}{\mathtt{p} \lhd P \parallel \overline{\mathtt{p}} \lhd Q \xrightarrow{\tau} \mathtt{p} \lhd P' \parallel \overline{\mathtt{p}} \lhd Q'}$$

$$\text{R-Rec}\ \frac{}{\mathtt{p} \lhd \mu\chi.P \parallel \overline{\mathtt{p}} \lhd Q \xrightarrow{\tau} \mathtt{p} \lhd P\{\mu\chi.P/\chi\} \parallel \overline{\mathtt{p}} \lhd Q}$$

$$\text{R-IfT}\ \frac{e \downarrow \mathsf{true}}{\mathtt{p} \lhd \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \parallel \overline{\mathtt{p}} \lhd R \xrightarrow{\tau} \mathtt{p} \lhd P \parallel \overline{\mathtt{p}} \lhd R}$$

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

Fig. 2. Labelled transition rules for sessions (we omit R-IfF)

2.2 Labelled Transition Semantics

We assume an *evaluation* (partial) function $\downarrow$ transforming expressions e into values v, written $e \downarrow v$.

The operational semantics of sessions are defined modulo a *structural congruence* relation over sessions $\mathcal{M}$, noted $\equiv \subseteq \mathbb{M} \times \mathbb{M}$.

$$\mathtt{p} \lhd P \parallel \overline{\mathtt{p}} \lhd Q \equiv \overline{\mathtt{p}} \lhd Q \parallel \mathtt{p} \lhd P$$

The *labelled transition rules* are defined in Figure 2; we just present the left rules. The rule have the form $\mathcal{M} \xrightarrow{\alpha} \mathcal{M}$, where $\alpha \in \mathcal{A}$ is an *action*.

$$\alpha := \mathtt{r}?l(v) \mid \mathtt{r}!l\langle v\rangle \mid \tau$$

A *computation* is a sequence of τ-transitions or *reductions* $\mathcal{M}_1 \xrightarrow{\tau} \mathcal{M}_2 \xrightarrow{\tau} \cdots$. 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 its dual $\overline{\mathtt{p}}$ on the label l can do a transition labelled by $\overline{\mathtt{p}}?l(v)$ 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 $\overline{\mathtt{p}}$ on label l an expression e that can be evaluated as v to do a transition labelled by $\overline{\mathtt{p}}!l\langle v\rangle$ 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 dual participants $\mathtt{p}$ and $\overline{\mathtt{p}}$ is performed by means of rule R-Com. Whenever $\mathtt{p} \lhd P$ can do a transition labelled by the input action $\overline{\mathtt{p}}?l(v)$ and reach the redex $\mathtt{p} \lhd P'$, and $\overline{\mathtt{p}} \lhd Q$ can do a transition labelled by the output action $\mathtt{p}!l\langle v\rangle$ and reach the redex $\overline{\mathtt{p}} \lhd Q'$, we can infer a τ transition from the composition of $\mathtt{p} \lhd P$ and $\overline{\mathtt{p}} \lhd Q$ to the composition of $\mathtt{p} \lhd P'$ and $\overline{\mathtt{p}} \lhd Q'$.

Rule R-Rec allows a participant $\mathtt{p}$ recursively defined as $\mu\chi.P$ and running in parallel with its dual $\overline{\mathtt{p}} \lhd Q$, 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 $\mathtt{p} \lhd P\{\mu\chi.P/\chi\} \parallel \overline{\mathtt{p}} \lhd Q$.

Rule R-IfT says that a participant $\mathtt{p}$ with the body $\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$ and running in parallel with the dual thread $\overline{\mathtt{p}} \lhd R$, can do a τ-transition and reach the redex $\mathtt{p} \lhd P \parallel \overline{\mathtt{p}} \lhd R$ whenever the expression e evaluates to true.

Rule R-Str rearranges processes with structural congruence.

Example 2. Consider the authorisation protocol in § 1 and let

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

where process $P_{\mathtt{c}}$ implements the client $\mathtt{c}$, and process $P^*_{\mathtt{a}} \stackrel{\text{def}}{=} Q_{\mathtt{a}}\{P_{\mathtt{a}}/\chi\}$ implements the (unfolding of the) authorisation server $\mathtt{a}$. Assume that $\overline{\mathtt{c}} = \mathtt{a}$: that is, the server and the client are dual participants.

We want to analyse a communication of the server $\mathtt{a}$ with the client $\mathtt{c}$ depicting a *password* transaction, that is a reduction inferred from the label $\mathtt{pwd}$. A first application of rule R-Rec unfolds the server $\mathtt{c}$:

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

where $P^*_{\mathtt{c}} \stackrel{\text{def}}{=} \mathtt{a}!\mathsf{pwd}\langle\text{“fido”}\rangle.P_{\mathtt{c}} + \mathtt{a}!\mathsf{quit}\langle\rangle$.

Now we apply rule R-Com to infer a communication among the server $\mathtt{a}$ and the client $\mathtt{c}$ on the label $\mathtt{pwd}$, followed by R-Str:

$$\text{R-Str}\ \frac{\text{R-Com}\ \dfrac{(A) \qquad (B)}{\mathtt{a} \lhd P^*_{\mathtt{a}} \parallel \mathtt{c} \lhd P^*_{\mathtt{c}} \stackrel{\tau}{\longrightarrow} \mathtt{a} \lhd P_{\mathtt{a}} \parallel \mathtt{c} \lhd P_{\mathtt{c}}}}{\mathcal{M}^* \stackrel{\tau}{\longrightarrow} \mathtt{c} \lhd P_{\mathtt{c}} \parallel \mathtt{a} \lhd P_{\mathtt{a}}}$$

where

$$\text{R-Sum-L}\ \frac{\text{R-Inp}\ \dfrac{}{\mathtt{a} \lhd \mathtt{c}?\mathsf{pwd}(x).P_{\mathtt{a}} \xrightarrow{\mathtt{c}?\mathtt{pwd}(\text{“fido”})} \mathtt{a} \lhd P_{\mathtt{a}}}}{(A)\ \mathtt{a} \lhd P^*_{\mathtt{a}} \xrightarrow{\mathtt{c}?\mathtt{pwd}(\text{“fido”})} \mathtt{a} \lhd P_{\mathtt{a}}}$$

and

$$\text{R-Sum-L}\;\frac{\text{R-Out}\;\dfrac{}{\mathsf{c} \lhd \mathsf{a}!\mathsf{pwd}\langle\text{“fido”}\rangle.P_{\mathsf{c}} \xrightarrow{\mathsf{a}!\mathsf{pwd}\langle\text{“fido”}\rangle} \mathsf{c} \lhd P_{\mathsf{c}}}}{(B)\;\mathsf{c} \lhd P_{\mathsf{c}}^{*} \xrightarrow{\mathsf{a}!\mathsf{pwd}\langle\text{“fido”}\rangle} \mathsf{c} \lhd P_{\mathsf{c}}}$$

2.3 Mechanisation and Implementation of Types and Processes

The mechanisation of types and processes in Rocq [46] relies on the command `Inductive`. As an example, we show the mechanisation of iso-recursive session types (cf. types T of Definition 1). In the code below, `var` is notation for `nat`, `participant` and `label` are types built using seeds of type `nat`, and `styp` implements sorts S of Definition 1.

```
Inductive typ : Set :=
| typ_var : var → typ
| typ_mu : var → typ → typ
| typ_sum : typ → typ → typ
| typ_input : participant → label → styp → typ → typ
| typ_output : participant → label → styp → typ → typ
| typ_end : typ
| typ_bot : typ.
```

The mechanisation of the syntax of processes is similar. The semantics of sessions of Figure 2 are instrumented with two technical devices to simplify the proof of subject reduction:

(i) We distinguish among τ-actions originating from unfolding (cf. R-Rec) and from if-then-else (e.g. R-IfT);
(ii) We shift the bound identifiers of the recursive processes $\mu\chi.P$ that substitutes process variable χ in the continuation of R-Rec.

Item (ii) allows to maintain a *freshness invariant*: to avoid using the same identifier twice in binders. That is, we require that all variables x and process variables χ occurring in processes $\mathtt{r}?l(x).P$ and $\mu X.P$, respectively, are syntactically distinct (cf. § 4.3).

The implementation of rule R-Rec below relies on a natural number M bigger than the maximum seed (cf. `max_seed`) used in the (bound) identifiers of the process, and on function `shiftP` shifting all identifiers used in bound variables and bound process variables of M positions. Action `aunfold` implementes the τ-action for unfolding (cf. (i)).

```
Check r_rec: ∀ (p : participant ) (X : process_var ) (P Q RP : process ) (M : nat),
   max_seed (proc_mu X P) < M → RP = shiftP M (proc_mu X P) →
   lts (parallel_session (p, proc_mu X P) (pdual p, Q)) aunfold
   (parallel_session (p, substP P RP X) (pdual p, Q)).
```

Implementation Types and processes can be exported to functional languages as *algebraic data types* by using the command `Extraction`. We note that the labelled transition semantics of Figure 2 are mechanised by using the command `Inductive`, and in turn cannot be executed. However, we could easily define `Fixpoint`-based

semantics by identifying a termination measure (cf. [25,26]) to provide for a certified interpreter that can be extracted e.g. to OCaml and ran. Alternatively, we could compile processes into Go programs [20] and run them.

3 Type Congruence

In this section, we introduce type congruence and explain its Rocq mechanisation.

3.1 Type Congruence

To recover the flexibility of type systems based on equi-recursive types, e.g. [30,17,23], we define a notion of *type congruence* (noted $\equiv$) relating folded and unfolded types, and use it at top level when composing two sessions (cf. § 4). Following the approach pioneered by [37, Chapter 4], we let type congruence be the union of all symmetric equivalences.

We devise a notion of type equivalence that is tailored at equating folded and unfolded *well-formed* types (cf. Definition 2 and § 2.1).

Definition 3 (Type Equivalence). *A relation $\mathcal{R} \subseteq \mathcal{T} \times \mathcal{T}$ is a* type equivalence *whenever $T_1 \mathcal{R} T_2$ and*

- $T_1 = \mu X.U_1$ *and* $T_2 = \mu X.U_2$ *imply* $U_1 \mathcal{R} U_2$ *and* $U_1\{\mu X.U_1/X\} \mathcal{R} T_2$ *and* $T_1 \mathcal{R} U_2\{\mu X.U_2/X\}$
- $T_1 = \mu X.U_1$ *and* $T_2 \neq \mu X.U_2$ *imply* $U_1\{\mu X.U_1/X\} \mathcal{R} T_2$
- $T_1 \neq \mu X.U_1$ *and* $T_2 = \mu X.U_2$ *imply* $T_1 \mathcal{R} U_2\{\mu X.U_2/X\}$
- $T_1 = X_1$ *and* $T_2 = X_2$
- $T_1 = \mathsf{end}$ *and* $T_2 = \mathsf{end}$
- $T_1 = \mathtt{p}?l(S).U_1 + R_1$ *and* $T_2 = \mathtt{p}?l(S).U_2 + R_2$ *imply* $U_1 \mathcal{R} U_2$ *and* $R_1 \mathcal{R} R_2$
- $T_1 = \mathtt{p}!l(S).U_1 + R_1$ *and* $T_2 = \mathtt{p}!l(S).U_2 + R_2$ *imply* $U_1 \mathcal{R} U_2$ *and* $R_1 \mathcal{R} R_2$
- $T_1 = \mathtt{p}?l(S).U_1$ *and* $T_2 = \mathtt{p}?l(S).U_2$ *imply* $U_1 \mathcal{R} U_2$
- $T_1 = \mathtt{p}!l(S).U_1$ *and* $T_2 = \mathtt{p}!l(S).U_2$ *imply* $U_1 \mathcal{R} U_2$
- T_1 *and* T_2 *do not match the cases above imply* False.

Definition 4 (Structural Equivalence). *A relation $\mathcal{R} \subseteq \mathcal{T} \times \mathcal{T}$ is a* structural equivalence *whenever it is (i) a type equivalence and (ii) symmetric.*

Definition 5 (Type Congruence). Type congruence*, noted $\equiv \subseteq \mathcal{T} \times \mathcal{T}$, is the union of all structural equivalences.*

3.2 Mechanisation of Type Congruence

In this section, we mechanise type congruence (cf. Definition 5) in Rocq. The objective is to obtain a computer aided proof of two results (cf. Lemma 5 and Lemma 6) used to close subject reduction. We also showcase a method to prove that two types are congruent (cf. Example 3).

```
1  Notation X £ T := (substT T X (typ_mu X T)) (at level 40).
2  Definition equiv R := ∀ (t1 t2 : typ), R t1 t2 →
3      match t1, t2 with
4      | typ_mu X1 U1, typ_mu X2 U2 ⇒ R U1 U2 ∧ R (X1 £ U1) t2 ∧ R t1 (X2 £ U2)
5      | typ_mu X U, _ ⇒ R (X £ U) t2
6      | _, typ_mu X U ⇒ R t1 (X £ U)
7      | typ_var X1, typ_var X2 ⇒ True
8      | typ_end, typ_end ⇒ True
9      | typ_sum (typ_input p1 l1 s1 u1) r1, typ_sum (typ_input p2 l2 s2 u2) r2 ⇒
10       p1 = p2 ∧ l1 = l2 ∧ s1 = s2 ∧ R u1 u2 ∧ R r1 r2
11     | typ_sum (typ_output p1 l1 s1 u1) r1, typ_sum (typ_output p2 l2 s2 u2) r2 ⇒
12        p1 = p2 ∧ l1 = l2 ∧ s1 = s2 ∧ R u1 u2 ∧ R r1 r2
13     | typ_input p1 l1 s1 u1, typ_input p2 l2 s2 u2 ⇒
14        p1 = p2 ∧ l1 = l2 ∧ s1 = s2 ∧ R u1 u2
15     | typ_output p1 l1 s1 u1, typ_output p2 l2 s2 u2 ⇒
16        p1 = p2 ∧ l1 = l2 ∧ s1 = s2 ∧ R u1 u2
17     | _, _ ⇒ False
18     end.
19 Definition struct_equiv R := equiv R ∧ symmetric typ R.
20 Definition relation2 (X Y: Type) := X → Y → Prop.
21 Definition set(X: Type) := X → Prop.
22 Section O.
23   Variables X Y: Type.
24   Definition union_st (P: set (relation2 X Y)) := fun x y ⇒ exists2 R, P R & R x y.
25 End O.
26 Definition typ_scongr := union_st typ typ struct_equiv .
27 Notation "T1 == T2" := (typ_scongr T1 T2) (at level 40).
28 Lemma equiv_scongr : equiv typ_scongr .
```

Fig. 3. Mechanisation of type congruence in Rocq

The mechanisation is in Figure 3. The predicate `equiv` on binary relations over `typ` implements Definition 3. The key cases for iso-recursion are lines 4–6: in order for a relation $\mathcal{R}$ to satisfy `equiv`, it must be that whenever a folded type $\mu X.T$ is related to U, then its unfolding $T\{\mu X.T/X\}$ is also related to U. The remaining cases are homomorphic.

The predicate `struct_equiv` implements Definition 4 and holds for relations $\mathcal{R}$ that satisfy `equiv` and are symmetric. The relation `typ_scongr` mechanises $\equiv$ and is defined as the union of all relations $\mathcal{R}$ that satisfy `struct_equiv` . The operator `union_st` implementing this abstraction is defined in lines 20–25, which are taken from the formalisation of [45]. The notation `exists2 R, P x & Q x` expresses the existence of R which satisfies both predicates P and Q.[1]

The lemma in line 28 mechanises the following result.

Lemma 1. *Type congruence is a type equivalence.*

Example 3. Take a participant q and labels l_1, l_2, and consider the types T_1 and T_2 defined below. Let i and b be short for int and bool, respectively. We have that $T_1 \equiv T_2$.

$$T_1 \stackrel{\text{def}}{=} \mu X.(\mathsf{q}?l_1(\mathsf{b}).\mathsf{q}!l_1(\mathsf{i}).X + U_1) \qquad U_1 \stackrel{\text{def}}{=} \mathsf{q}?l_2(\mathsf{i}).\mathsf{q}!l_2(\mathsf{i}).\mathsf{q}!l_2(\mathsf{i}).\mu Y.\mathsf{q}!l_2(\mathsf{i}).Y$$

$$T_2 \stackrel{\text{def}}{=} \mathsf{q}?l_1(\mathsf{b}).\mathsf{q}!l_1(\mathsf{i}).T_1 + U_2 \qquad U_2 \stackrel{\text{def}}{=} \mathsf{q}?l_2(\mathsf{i}).\mu Y.\mathsf{q}!l_2(\mathsf{i}).Y$$

[1] `https://rocq-prover.org/doc/v8.20/stdlib/Coq.Init.Logic.html`

$$\text{S-Nat}\ \frac{n \in \mathbb{N}}{\Gamma \vdash n\colon \mathsf{nat}} \qquad \text{S-Int}\ \frac{z \in \mathbb{Z}}{\Gamma \vdash z\colon \mathsf{int}} \qquad \text{S-Str-E}\ \frac{}{\Gamma \vdash ""\colon \mathsf{str}}$$

$$\text{S-Str}\ \frac{c \in \mathrm{Char} \quad \Gamma \vdash s\colon \mathsf{str}}{\Gamma \vdash c\hat{}s\colon \mathsf{str}} \qquad \text{S-Bl}\ \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash b\colon \mathsf{bool}} \qquad \text{S-Uni}\ \frac{}{\Gamma \vdash ()\colon \mathsf{unit}}$$

$$\text{S-Var}\ \frac{}{\Gamma, x\colon S \vdash x\colon S} \qquad \text{S-Lt}\ \frac{\Gamma \vdash e_1\colon S \quad \Gamma \vdash e_2\colon S \quad S \in \{\mathsf{nat}, \mathsf{int}\}}{\Gamma \vdash e_1 < e_2\colon \mathsf{bool}}$$

$$\text{S-Eq}\ \frac{\Gamma \vdash e_1\colon S \quad \Gamma \vdash e_2\colon S}{\Gamma \vdash e_1 = e_2\colon \mathsf{bool}} \qquad \text{S-Neg}\ \frac{\Gamma \vdash e\colon \mathsf{bool}}{\Gamma \vdash \neg e\colon \mathsf{bool}}$$

$$\text{S-And}\ \frac{\Gamma \vdash e_1\colon \mathsf{bool} \quad \Gamma \vdash e_2\colon \mathsf{bool}}{\Gamma \vdash e_1 \wedge e_2\colon \mathsf{bool}} \qquad \text{S-Or}\ \frac{\Gamma \vdash e_1\colon \mathsf{bool} \quad \Gamma \vdash e_2\colon \mathsf{bool}}{\Gamma \vdash e_1 \vee e_2\colon \mathsf{bool}}$$

Fig. 4. Sorting rules

To prove this claim we provide a relation $\mathcal{R}$ that satisfies `struct_equiv` and s.t. $T_1 \mathcal{R} T_2$. In order to systematically use this methodology, we deploy a proof technique in the tactic language `Ltac`.

```
Lemma exist_struct_equiv : ∀ (R : typ → typ → Prop) (t1 t2 : typ),
  struct_equiv R → R t1 t2 → t1 == t2.
Ltac prove_scongr R := match goal with | ⊢ ?W1 == ?W2 ⇒
  eapply (exist_struct_equiv R); eauto end.
```

4 Type System

The sorting rules used in the typing system are presented in Figure 4.

Type environments Γ map variables to sorts and process variables to types:

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

We first present the rules for values v. Rule S-Nat allows for typing a number n in the set of naturals $\mathbb{N}$ with type nat. Rule S-Int is used to type a number z in the set of integers $\mathbb{Z}$ with type int. Rule S-Str-E type-checks the empty string. Non-empty strings are typed by concatenating characters c with strings s, written $c\hat{}s$, by means of rule S-Str. Boolean constants are typed with rule S-Bl. The only value of type unit is the empty value, noted (), and is typed by using rule S-Uni.

Variables x are typed with rule S-Var.

Expressions e are typed with the remaining rules. Rule S-Lt accepts a "less than" test among expressions e_1 and e_2 whenever e_1 and e_2 have the same numerical sort. Rule S-Eq does an equality test among expressions e_1 and e_2 of the same sort. Rule S-Neg is used to negate a boolean expression e. Rule S-And type-checks a conjunction of two boolean expressions e_1 and e_2. Rule S-Or accepts a disjunction of two boolean expressions e_1 and e_2.

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

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

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

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

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

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

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

$$\text{T-Thr}\ \frac{\Gamma \vdash P : T \qquad \mathsf{wf}(T)}{\Gamma \Vdash \mathtt{p} \lhd P : T}$$

$$\text{T-Sess}\ \frac{\Gamma \Vdash \mathtt{p} \lhd P : T_1 \qquad \Gamma \Vdash \overline{\mathtt{p}} \lhd Q : T_2 \qquad \overline{T_1} \equiv T_2}{\Gamma \Vdash \mathtt{p} \lhd P \parallel \overline{\mathtt{p}} \lhd Q : \bot}$$

Fig. 5. Type system

4.1 Typing Processes

The type system is defined in Figure 5; in this section we analyse the rules for processes, while § 4.2 is devoted to the rules for sessions.

Typing judgements for processes have the form $\Gamma \vdash P : T$, while typing judgements for sessions have the form $\Gamma \Vdash \mathcal{M} : T$ 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. § 2.1): 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 session types is T-Rec. In order to allow Γ to type a recursion process $\mu\chi.P$ with a type $\mu X.T$, we require that $\Gamma, \chi : \mu X.T$ 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. [18], where the type of $\mu\chi.P$ and the type of the 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 $\Gamma, \chi : \mu X.T$ 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.

$$
\begin{aligned}
U_{\mathtt{c}} &\stackrel{\text{def}}{=} \mu X.(\mathtt{a}!\mathsf{pwd}(\mathsf{str}).(\mathtt{a}?\mathsf{ok}(\mathsf{bool}).\chi + \mathtt{a}?\mathsf{fail}(\mathsf{unit}).\mathsf{end}) + \mathtt{a}!\mathsf{quit}(\mathsf{unit}).\mathsf{end})\\
U_{\mathtt{a}} &\stackrel{\text{def}}{=} \mu X.(\mathtt{c}?\mathsf{pwd}(\mathsf{str}).(\mathtt{c}!\mathsf{ok}(\mathsf{bool}).\chi + \mathtt{c}!\mathsf{fail}(\mathsf{unit}).\mathsf{end}) + \mathtt{c}?\mathsf{quit}(\mathsf{unit}).\mathsf{end})\\
Q_{\mathtt{c}} &\stackrel{\text{def}}{=} \mu\chi.(\mathtt{a}!\mathsf{pwd}\langle\text{“fido”}\rangle.(\mathtt{a}?\mathsf{ok}(x).\chi + \mathtt{a}?\mathsf{fail}(x)) + \mathtt{a}!\mathsf{quit}\langle\rangle)\\
Q_{\mathtt{a}} &\stackrel{\text{def}}{=} \mu\chi.(\mathtt{c}?\mathsf{pwd}(x).\mathit{Check}_{\mathtt{a}} + \mathtt{c}?\mathsf{quit}(x))\\
\mathit{Check}_{\mathtt{a}} &\stackrel{\text{def}}{=} \mathsf{if}\ x = \text{“miau”}\ \mathsf{then}\ \mathtt{c}!\mathsf{ok}\langle\mathsf{true}\rangle.\chi\ \mathsf{else}\ \mathtt{c}!\mathsf{fail}\langle\rangle
\end{aligned}
$$

Fig. 6. Variant of authorisation protocol in § 1

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 without duplicated labels, or output types without duplicated labels, respectively (cf. § 2.1). 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 branching 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 $\&_{i\in 1,\ldots,n+1}\mathtt{r}?l_i(S).T_i$ (cf. Definition 2). 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 4. Consider the variant of the authentication protocol of Figure 6, where $U_{\mathtt{c}}, U_{\mathtt{a}}$ are the types of the client $\mathtt{c}$ and of the authorisation server $\mathtt{a}$, respectively, and $Q_{\mathtt{c}}, Q_{\mathtt{a}}$ are process implementations of the client $\mathtt{c}$ and of the authorisation server $\mathtt{a}$, respectively. The semantics of the protocol are better described by the types $U_{\mathtt{c}}$ and $U_{\mathtt{a}}$. Type $U_{\mathtt{c}}$ prescribes that, as in the protocol of § 1, the client $\mathtt{c}$ sends a password to the server $\mathtt{a}$, or it quits. Differently from § 1, the client $\mathtt{c}$ after sending the password waits for an ack from the server $\mathtt{a}$: the server can send *ok* and the protocol reinitiates, or it can send *fail* and the protocol ends. The behaviour of the server $\mathtt{a}$ described by its type $U_{\mathtt{a}}$ is *dual* to the one of the client $\mathtt{c}$ (cf. Definition 6).

Consider now the implementations Q_{c} and Q_{a} of the client c and of the server a, respectively. Process Q_{c} mirrors the behaviour of its type U_{c}. Process Q_{a} presents an interesting behaviour based on the *check* of the password received by the client c: if the password is correct, then Q_{a} sends an *ok* ack and the protocol restarts, else it sends a *fail* ack and the protocol ends.

Based on the rules for processes of Figure 5, we have the following typings:

(i) $\emptyset \vdash Q_{\mathsf{c}} : U_{\mathsf{c}}$;
(ii) $\emptyset \vdash Q_{\mathsf{a}} : U_{\mathsf{a}}$.

We discuss the typing of (ii), which is relevant to show that process $\mathit{Check}_{\mathsf{a}}$ is well-typed.

The type inference (ii) is obtained by applying T-Rec to the hypothesis

$$\chi : U_{\mathsf{a}} \vdash \mathsf{c}?\mathsf{pwd}(x).\mathit{Check}_{\mathsf{a}} + \mathsf{c}?\mathsf{quit}(x) : \mathsf{c}?\mathsf{pwd}(\mathsf{str}).C_{\mathsf{a}} + \mathsf{c}?\mathsf{quit}(\mathsf{unit}).\mathsf{end} \quad (1)$$
$$C_{\mathsf{a}} \stackrel{\text{def}}{=} \mathsf{c}!\mathsf{ok}(\mathsf{bool}).U_{\mathsf{a}} + \mathsf{c}!\mathsf{fail}(\mathsf{unit}).\mathsf{end}$$

To type check (1), we apply T-Sum to the hypotheses

$$\chi : U_{\mathsf{a}} \vdash \mathsf{c}?\mathsf{pwd}(x).\mathit{Check}_{\mathsf{a}} : \mathsf{c}?\mathsf{pwd}(\mathsf{str}).C_{\mathsf{a}} \quad (2)$$
$$\chi : U_{\mathsf{a}} \vdash \mathsf{c}?\mathsf{quit}(x) : \mathsf{c}?\mathsf{quit}(\mathsf{unit}).\mathsf{end} \quad (3)$$

The item (3) is discharged by using T-Inp, followed by T-End. To tackle (2), we apply T-Inp to the hypothesis below, where $\Gamma \stackrel{\text{def}}{=} \chi : U_{\mathsf{a}}, x : \mathsf{str}$:

$$\Gamma \vdash \mathit{Check}_{\mathsf{a}} : C_{\mathsf{a}} \quad (4)$$

Item (4) is obtained by an application of T-If to the hypotheses

$$\Gamma \vdash \mathsf{c}!\mathsf{ok}\langle\mathsf{true}\rangle.\chi : C_{\mathsf{a}} \quad (5)$$
$$\Gamma \vdash \mathsf{c}!\mathsf{fail}\langle\rangle : C_{\mathsf{a}} \quad (6)$$

To type check (5) we apply T-Sum-L to the hypothesis

$$\Gamma \vdash \mathsf{c}!\mathsf{ok}\langle\mathsf{true}\rangle.\chi : \mathsf{c}!\mathsf{ok}(\mathsf{bool}).U_{\mathsf{a}} \quad (7)$$

while to type check (6) we apply T-Sum-R to the hypothesis

$$\Gamma \vdash \mathsf{c}!\mathsf{fail}\langle\rangle : \mathsf{c}!\mathsf{fail}(\mathsf{unit}).\mathsf{end} \quad (8)$$

Item 7 is proved by applying T-Out, followed by T-Var. Item 8 is a consequence of applying T-Out, followed by T-End.

4.2 Type Checking Sessions

The typing rules for sessions of Figure 5 have the form $\Gamma \Vdash \mathcal{M} : T$ and use the rules for processes $\Gamma \vdash P : T$. 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. Figure 1), then the type system $\Vdash$ assigns the type T to the thread $\mathtt{p} \triangleleft P$.

Rule T-Sess is used to type *a pair of threads*, and it relies on *type congruence* (cf. § 3) and on *type duality*. Since we consider iso-recursive types, we define duality as a terminating function rather than as a coinductive relation (cf. [17]).

Definition 6 (Type Duality). *Function* $\overline{\cdot} : \mathcal{T} \to \mathcal{T}$ *is defined by the following patterns:*

$$\overline{\mathtt{r}!l(S).T} \stackrel{def}{=} \overline{\mathtt{r}}?l(S).\overline{T} \qquad \overline{\mathtt{r}?l(S).T} \stackrel{def}{=} \overline{\mathtt{r}}!l(S).\overline{T} \qquad \overline{\mu X.T} \stackrel{def}{=} \mu X.\overline{T}$$

$$\overline{T_1 + T_2} \stackrel{def}{=} \overline{T_1} + \overline{T_2} \qquad \overline{T} \stackrel{def}{=} T \ (else)$$

Lemma 2 (Termination). *Type duality is a terminating function.*

The recursive definition is accepted by Rocq because it is "guarded on 1st argument". The involution `pdual` implements duality of participants.

```
Function typ_dual (t : typ) : typ :=
  match t with
  | typ_mu x t0 ⇒ typ_mu x (typ_dual t0)
  | typ_sum t1 t2 ⇒ typ_sum (typ_dual t1) (typ_dual t2)
  | typ_input p l s t0 ⇒ typ_output (pdual p) l s (typ_dual t0)
  | typ_output p l s t0 ⇒ typ_input (pdual p) l s (typ_dual t0)
  | _ ⇒ t
  end.
```

Lemma 3 (Involution). *Type duality is an involution.*

Lemma 4 (Injectivity). *Type duality is injective.*

These results are mechanised by structural induction on the parameters.

```
Lemma typ_dual_involution : ∀ t : typ, typ_dual (typ_dual t) = t.
Lemma dual_injective : ∀ t1 t2 : typ, typ_dual t1 = typ_dual t2 → t1 = t2.
```

Example 5. Consider types U_{c} and U_{a} of Figure 6. By applying the definition, we have $\overline{U_{\mathsf{c}}} = U_{\mathsf{a}}$. The result can be obtained by using the function introduced in the proof of Lemma 2 and by applying the tactic `repeat simpl`.

The rule for composing threads says that if a session is composed with a participant and its dual, and those are typed respectively with T_1 and T_2, and there is a *proof* that $\overline{T}_1 \equiv T_2$ (cf. Example 3), then the session is typed with type $\bot$. As outlined in § 2, the rule can be extended to multiple pairs of dual participants.

Example 6. We show an example of application of rule T-Sess. Consider the OAuth2 protocol of § 1, which we recap here for convenience by letting $s \stackrel{\text{def}}{=}$ "fido", s be short for str, and u be short for unit. Remember that $\overline{\mathsf{c}} = \mathsf{a}$.

$$T_{\mathsf{c}} \stackrel{\text{def}}{=} \mu X.(\mathsf{a}!\mathsf{pwd}(\mathsf{s}).X + \mathsf{a}!\mathsf{quit}(\mathsf{u}).\mathsf{end}) \qquad T_{\mathsf{a}} \stackrel{\text{def}}{=} \mu X.R_{\mathsf{a}}$$

$$R_{\mathsf{a}} \stackrel{\text{def}}{=} \mathsf{c}?\mathsf{pwd}(\mathsf{s}).X + \mathsf{c}?\mathsf{quit}(\mathsf{u}).\mathsf{end} \qquad T_{\mathsf{a}}^{*} \stackrel{\text{def}}{=} R_{\mathsf{a}}\{T_{\mathsf{a}}/X\}$$

$$P_{\mathsf{c}} \stackrel{\text{def}}{=} \mu\chi.(\mathsf{a}!\mathsf{pwd}\langle s\rangle.\chi + \mathsf{a}!\mathsf{quit}\langle\rangle) \qquad P_{\mathsf{a}} \stackrel{\text{def}}{=} \mu\chi.Q_{\mathsf{a}}$$

$$Q_{\mathsf{a}} \stackrel{\text{def}}{=} \mathsf{c}?\mathsf{pwd}(x).\chi + \mathsf{c}?\mathsf{quit}(x) \qquad P_{\mathsf{a}}^{*} \stackrel{\text{def}}{=} Q_{\mathsf{a}}\{P_{\mathsf{a}}/\chi\}$$

To illustrate the folding/unfolding mechanism while typing sessions, we show that the composition of $\mathsf{c} \triangleleft P_\mathsf{c}$ and $\mathsf{a} \triangleleft P^*_\mathsf{a}$ is well-typed.

In the following, we prove that the folded process P_c type-checks with the folded type T_c, while the unfolded process P^*_a type checks with the unfolded type T^*_a. We note that these choices of folding and unfolding types are indeed mandatory. That is, iso-recursive session types do not allow to type-check e.g. P^*_a with T_a, or P_a with T^*_a: this is in contrast with equi-recursive session types, e.g. [18], where T_a and T^*_a are equal, and can thus be exchanged at any time.

We start by assigning the type T_c to the client process P_c.

$$\text{T-Rec}\,\frac{\text{T-Sum}\,\dfrac{(A) \qquad (B)}{\chi\colon T_\mathsf{c} \vdash \mathsf{a}!\mathsf{pwd}\langle s\rangle.\chi + \mathsf{a}!\mathsf{quit}\langle\rangle\colon \mathsf{a}!\mathsf{pwd}(\mathsf{s}).T_\mathsf{c} + \mathsf{a}!\mathsf{quit}(\mathsf{u}).\mathsf{end}}}{\emptyset \vdash P_\mathsf{c}\colon T_\mathsf{c}}$$

$$\text{T-Out}\,\frac{\text{T-Var}\,\dfrac{}{\chi\colon T_\mathsf{c} \vdash \chi\colon T_\mathsf{c}}}{(A)\ \chi\colon T_\mathsf{c} \vdash \mathsf{a}!\mathsf{pwd}\langle s\rangle.\chi\colon \mathsf{a}!\mathsf{pwd}(\mathsf{s}).T_\mathsf{c}}$$

$$\text{T-Out}\,\frac{\text{T-End}\,\dfrac{}{\chi\colon T_\mathsf{c} \vdash \mathbf{0}\colon \mathsf{end}}}{(B)\ \chi\colon T_\mathsf{c} \vdash \mathsf{a}!\mathsf{quit}\langle\rangle\colon \mathsf{a}!\mathsf{quit}(\mathsf{u}).\mathsf{end}}$$

Next we assign the type T^*_a to the unfolded server P^*_a. Let $\Gamma \stackrel{\text{def}}{=} x\colon \mathsf{s}, \chi\colon T_\mathsf{a}$.

$$\text{T-Sum}\,\frac{(C) \qquad (D)}{\emptyset \vdash P^*_\mathsf{a}\colon \mathsf{c}?\mathsf{pwd}(\mathsf{s}).T_\mathsf{a} + \mathsf{c}?\mathsf{quit}(\mathsf{u}).\mathsf{end}}$$

$$\text{T-Inp}\,\frac{\text{T-End}\,\dfrac{}{x\colon \mathsf{u} \vdash \mathbf{0}\colon \mathsf{end}}}{(D)\ \emptyset \vdash \mathsf{c}?\mathsf{quit}(x)\colon \mathsf{c}?\mathsf{quit}(\mathsf{u}).\mathsf{end}}$$

$$\text{T-Inp}\,\frac{\text{T-Rec}\,\dfrac{\text{T-Sum}\,\dfrac{(E) \qquad (F)}{\Gamma \vdash Q_\mathsf{a}\colon T^*_\mathsf{a}}}{x\colon \mathsf{s} \vdash P_\mathsf{a}\colon T_\mathsf{a}}}{(C)\ \emptyset \vdash \mathsf{c}?\mathsf{pwd}(x).P_\mathsf{a}\colon \mathsf{c}?\mathsf{pwd}(\mathsf{s}).T_\mathsf{a}}$$

$$\text{T-Inp}\,\frac{\text{T-Var}\,\dfrac{}{\Gamma, x'\colon \mathsf{s} \vdash \chi\colon T_\mathsf{a}}}{(E)\ \Gamma \vdash \mathsf{c}?\mathsf{pwd}(x).\chi\colon \mathsf{c}?\mathsf{pwd}(\mathsf{s}).T_\mathsf{a}}$$

$$\text{T-Inp}\,\frac{\text{T-End}\,\dfrac{}{\Gamma, x'\colon \mathsf{u} \vdash \mathbf{0}\colon \mathsf{end}}}{(F)\ \Gamma \vdash \mathsf{c}?\mathsf{quit}(x)\colon \mathsf{c}?\mathsf{quit}(\mathsf{u}).\mathsf{end}}$$

We note that both T_c and T^*_a are well-formed (cf. § 2). An application of T-Thr to both $\emptyset \vdash P_\mathsf{c}\colon T_\mathsf{c}$ and $\emptyset \vdash P^*_\mathsf{a}\colon T^*_\mathsf{a}$ let us infer:

$$\emptyset \Vdash \mathsf{c} \triangleleft P_\mathsf{c}\colon T_\mathsf{c} \tag{9}$$

$$\emptyset \Vdash \mathsf{a} \triangleleft P^*_\mathsf{a}\colon T^*_\mathsf{a} \tag{10}$$

To apply T-Sess and type the composition $\mathsf{c} \triangleleft P_{\mathsf{c}} \parallel \mathsf{a} \triangleleft P^*_{\mathsf{a}}$ we need to prove that $\overline{T_{\mathsf{c}}} \equiv T^*_{\mathsf{a}}$. Note that $\overline{T_{\mathsf{c}}} = T_{\mathsf{a}}$. We proceed as in Example 3 and find a relation $\mathcal{R}$ s.t. $T_{\mathsf{a}} \mathcal{R} T^*_{\mathsf{a}}$, and $\mathcal{R}$ is a structural equivalence. We can then conclude:

$$\text{T-Sess}\frac{(9) \qquad (10) \qquad \overline{T_{\mathsf{c}}} \equiv T^*_{\mathsf{a}}}{\emptyset \Vdash \mathsf{c} \triangleleft P_{\mathsf{c}} \parallel \mathsf{a} \triangleleft P^*_{\mathsf{a}} : \bot}$$

Example 7. To see an example of an untyped session, remember the type and process definitions in Example 6 and consider the type

$$T^{\#}_{\mathsf{a}} \stackrel{\text{def}}{=} \mathsf{c}?\mathsf{pwd}(\mathsf{s}).\mu X.\mathsf{c}?\mathsf{pwd}(\mathsf{s}).X + \mathsf{c}?\mathsf{quit}(\mathsf{u}).\mathsf{end}$$

and a server implementation $P^{\#}_{\mathsf{a}}$ typed with $T^{\#}_{\mathsf{a}}$. We cannot type $\mathsf{c} \triangleleft P_{\mathsf{c}} \parallel \mathsf{a} \triangleleft P^{\#}_{\mathsf{a}}$ because we cannot relate the dual of the type T_{c} of the client c with the type $T^{\#}_{\mathsf{a}}$ of the server a by means of type congruence.

To see that, we show that the hypothesis (ABS) leads to a contradiction.

$$\text{(ABS)} \qquad \overline{T_{\mathsf{c}}} \equiv T^{\#}_{\mathsf{a}}$$

We use the following mechanised results.

```
Lemma scongr_symmetric : symmetric typ typ_scongr.
Lemma scongr_comp : ∀ (X : var) (T U : typ), (@ X T) == U → (X £T) == U.
Lemma scongr_sum : ∀ (t1 t2 t3 t4 : typ),
  typ_sum t1 t2 == typ_sum t3 t4 → t1 == t3 ∧ t2 == t4.
Lemma scongr_input : ∀ (p : participant) (l : label) (s : styp) (t1 t2 : typ),
  (p ? l s . t1) == (p ? l s . t2) → t1 == t2.
Lemma scongr_input_sum_false : ∀ (p : participant) (l : label) (s : styp) (t t1 t2 : typ),
  (p ? l s . t) == typ_sum t1 t2 → False.
```

First, note that $\overline{T_{\mathsf{c}}} = \mu X.(\mathsf{c}?\mathsf{pwd}(\mathsf{s}).X + \mathsf{c}?\mathsf{quit}(\mathsf{u}).\mathsf{end}) = T_{\mathsf{a}}$. We apply `scongr_comp` to (ABS) and obtain

$$\mathsf{c}?\mathsf{pwd}(\mathsf{s}).T_{\mathsf{a}} + \mathsf{c}?\mathsf{quit}(\mathsf{u}).\mathsf{end} \equiv T^{\#}_{\mathsf{a}} \tag{11}$$

We apply `scongr_sum` to (11) and infer

$$\mathsf{c}?\mathsf{pwd}(\mathsf{s}).T_{\mathsf{a}} \equiv \mathsf{c}?\mathsf{pwd}(\mathsf{s}).\mu X.\mathsf{c}?\mathsf{pwd}(\mathsf{s}).X \tag{12}$$

We apply `scongr_input` to (12) and obtain

$$T_{\mathsf{a}} \equiv \mu X.\mathsf{c}?\mathsf{pwd}(\mathsf{s}).X \tag{13}$$

We apply `scongr_symmetric` , `scongr_comp` , `scongr_symmetric` to (13) and infer

$$T_{\mathsf{a}} \equiv \mathsf{c}?\mathsf{pwd}(\mathsf{s}).\mu X.\mathsf{c}?\mathsf{pwd}(\mathsf{s}).X \tag{14}$$

An application of `scongr_comp` to (14) let us infer

$$\mathsf{c}?\mathsf{pwd}(\mathsf{s}).T_{\mathsf{a}} + \mathsf{c}?\mathsf{quit}(\mathsf{u}).\mathsf{end} \equiv \mathsf{c}?\mathsf{pwd}(\mathsf{s}).\mu X.\mathsf{c}?\mathsf{pwd}(\mathsf{s}).X \tag{15}$$

We apply lemma `scongr_symmetric` , `scongr_input_sum_false` to (15) and obtain `False`, contradiction.

4.3 Mechanisation and Implementation of the Type System

The main purpose of the formalisation of the typing system in Rocq is to set the basis for the mechanisation of the Theorem of subject reduction (cf. Theorem 1). The mechanisation of the type system of Figure 5 relies on an `Inductive` predicate that follows the structure of the rules in the Figure, with one technical variation. To avoid issues related to the reuse of the same identifier in bound variables and bound process variables, we require that each *extension operation* $\Gamma, x\colon S$ (resp. $\Gamma, X\colon T$) carries a proof that $x \notin \mathrm{dom}(\Gamma)$ (resp. $X \notin \mathrm{dom}(\Gamma)$).

Specifically, we equip the rules in Figure 5 that make use of extension with an extra parameter carrying a proof that the extended variable is *fresh*. In the code below, `evar_beq` and `process_var_beq` are boolean equalities over variables and process variables, respectively.

```
Function extend {A B} (eqb : A → A → bool) (g : partial_fun A B) (x : A) (s : B)
  (pf : fresh g x) := fun y ⇒ if eqb x y then Some s else g y.
(** Extension of type environments *)
Notation "g ',,' x ':' t @ pf" := (extend evar_beq g x t pf) (at level 40).
(** Extension of process environments *)
Notation "h ',,' x '=:' t @ pf" := (extend process_var_beq h x t pf) (at level 40).
```

Note that the freshness assumption does not affect the expressiveness of the typing system, because we can always alpha-rename variables and process variables of processes before type checking.

Implementation of the type checker Besides the `Inductive`-based implementation of the type system of Figure 5 in Rocq, in related work [25,26] we implemented the typing rules for multiparty sessions in OCaml and automatically verified their soundness in Why3 (cf. [42,11,15]), thus making available a *certified type checker* for multiparty sessions. [26, App. I] contains all the details. Intuitively, the key idea is to rely on a fixed-point technique that allows *unfolding* iso-recursive session types *up-to a length n*: the parameter is given in input to the type checking function. This allows the termination of rule T-Rec of Figure 5. Specifically, in [25,26] we achieved decidable type checking by using in the rule for session composition (cf. T-Sess) a *terminating function* to assess the *compliance* of the multiparty session participants.

In order to deploy a decidable type checker for the system in Figure 5, we need to implement the proof-based formulation of T-Sess as a function, and in turn to devise a suitable notion of *duality up-to type congruence*. We envision that this concept can be deployed following the guideline of the implementation of function *cstep* in [26, App. I]: the code should be simpler, because of the presence of exactly two participants.

We plan to add to the repository [21] a folder containing the source code of a fully executable type checker implementing the system in Figure 5.

5 Proof of Subject Reduction

To attack the proof of subject reduction, we define a LTS of types. We will use the transition system to match the actions of processes, so it is practical to use the same actions α of the LTS of Figure 2.

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

$$\text{E-Out}\ \frac{\emptyset \vdash v : S}{\mathtt{r}!l(S).T \xrightarrow{\mathtt{r}!l\langle v\rangle} T} \qquad \text{E-In}\ \frac{\emptyset \vdash v : S}{\mathtt{r}?l(S).T \xrightarrow{\mathtt{r}?l(v)} T}$$

$$\text{E-Sel-L}\ \frac{T_1 \xrightarrow{\mathtt{r}!l\langle v\rangle} T'}{T_1 + T_2 \xrightarrow{\mathtt{r}!l\langle v\rangle} T'} \qquad \text{E-Bra-L}\ \frac{T_1 \xrightarrow{\mathtt{r}?l(v)} T'}{T_1 + T_2 \xrightarrow{\mathtt{r}?l(v)} T'}$$

$$\text{E-Rec}\ \frac{}{\mu X.T \xrightarrow{\tau} T\{\mu X.T/X\}} \qquad \text{E-Bot}\ \frac{}{\bot \xrightarrow{\tau} \bot}$$

Fig. 7. Labelled transition system of types

The left rules for types are in Figure 7. The rules are designed for well-formed types (cf. § 2.1), as we discuss below (cf. rules E-Sel-L, E-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.

Rule E-Rec allows a recursive type to execute a τ action and to unfold itself.

Rule E-Bot allows type $\bot$ to execute a τ action and reach itself.

5.1 Preliminary Lemmas

We now analyse the properties of type congruence (cf. Definition 5), and use these results in the proof of Theorem 1 (Subject Reduction). Some preliminary results on type congruence have been already introduced along the paper (cf. Lemma 1, Example 3, and Example 7).

A first result establishes that type congruence is preserved by duality.

Lemma 5. *If* $T_1 \equiv T_2$ *then* $\overline{T_1} \equiv \overline{T}_2$.

Remember that the notation == is short for `typ_scongr` (cf. Figure 3). The lemma is mechanised by using the `Ltac` tactic `prove_scongr` (cf. Example 3).

```
Lemma scongr_dual : ∀ (T1 T2 : typ), T1 == T2 → typ_dual T1 == typ_dual T2.
```

In order to introduce the next step towards subject reduction, we define the *duality predicate* between actions.

Definition 7 (Action duality). *The* action duality *predicate* $\mathcal{D} \subseteq \mathcal{A} \times \mathcal{A}$ *is defined by the following axioms:*

$$\frac{}{\mathtt{p}?l(v)\,\mathcal{D}\,\overline{\mathtt{p}}!l\langle v\rangle} \qquad \frac{}{\mathtt{p}!l\langle v\rangle\,\mathcal{D}\,\overline{\mathtt{p}}?l(v)}$$

Remind the notation $\mathsf{wf}(T)$ to indicate that T is well-formed (cf. § 2.1 and rule T-Thr of Figure 5). A key result is that type congruence is preserved by the LTS of Figure 7, in the following sense.

Lemma 6. *Assume that* $\mathsf{wf}(T_1)$ *and* $\mathsf{wf}(T_2)$*, and let* $T_1 \equiv \overline{T_2}$*. Let* $\alpha_1\,\mathcal{D}\,\alpha_2$*,* $T_1 \xrightarrow{\alpha_1} U_1$*, and* $T_2 \xrightarrow{\alpha_2} U_2$*. We have that* $U_1 \equiv \overline{U_2}$*.*

The lemma is mechanised by using dependent induction [36] on $T_1 \xrightarrow{\alpha_1} U_1$ and $T_2 \xrightarrow{\alpha_2} U_2$. In the following code, `typ_lts` corresponds to the transition system $T \xrightarrow{\alpha} T$ of Figure 7, while `action_dual` mechanises Definition 7.

```
Lemma elts_dual_scongr : ∀ (T1 T2 : typ) (a : proc_action ) (U1 U2 : typ) (b : proc_action ),
  wf T1 → wf T2 → T1 == typ_dual T2 → typ_lts T1 a U1 → typ_lts T2 b U2 →
  action_dual a b → U1 == typ_dual U2.
```

Next, we establish further results used in the proof of Theorem 1, that are type preservation of structural congruence, and of value and process substitution.

Lemma 7. *If* $\Gamma \Vdash \mathcal{M}_1 : T$ *and* $\mathcal{M}_1 \equiv \mathcal{M}_2$ *then* $\Gamma \Vdash \mathcal{M}_2 : T$*.*

The mechanisation of Lemma 7 simply follows by inversion of $\Gamma \Vdash \mathcal{M}_1 : T$. The predicate `types_session G K M U` corresponds to the judgement $\Gamma, \mathcal{K} \Vdash \mathcal{M} : U$ of Figure 5, where Γ is used for variables and $\mathcal{K}$ is used for process variables. The relation `proc_equiv` implements $\equiv$.

```
Lemma types_struct : ∀ (G : typ_env ) (H : proc_env ) (M1 M2 : session ) (U : typ),
  types_session G H M1 U → proc_equiv M1 M2 → types_session G H M2 U.
```

Lemma 8 (Value Substitution). *Assume that the following hold:*

1. $\Gamma, x : S \vdash P : U$*;*
2. $\Gamma \vdash v : S$*.*

We have that $\Gamma \vdash P\{v/x\} : U$*.*

The proof proceeds by induction on the judgement for processes. The predicate `stypes G e s` implements the judgement $\Gamma \vdash e : S$ (cf. Figure 4). Process `substV P v x` deploys $P\{v/x\}$.

```
Lemma substitution : ∀ (G : typ_env ) (H : proc_env ) (x : var) (s : styp) (P : process )
  (t : typ) (v : value ) (pf : fresh G x):
  types (G,, x —: s @ pf) H P t → stypes G (expr_val v) s → types G H (substV P v x) t.
```

Lemma 9 (Process Substitution). *Assume that the following hold:*

1. $\Gamma, \chi : T \vdash P : U$*;*
2. $\Gamma \vdash Q : T$*.*

We have that $\Gamma \vdash P\{Q/\chi\} : U$.

Process substitution is mechanised by applying structural induction to P. The predicate `types G K P U` corresponds to the judgement $\Gamma, \mathcal{K} \vdash P : U$ of Figure 5. Predicate `closedP` corresponds to notion of closed processes, function `bv` (resp. `bvP`) returns the bound variables (resp. bound process variables) of process P, and predicate `disjoint` holds when the lists received as parameters do not share any element. Process `substP P Q X` is $P\{Q/\chi\}$.

```
Lemma types_substP : ∀ (P Q : process) (T U : typ) (G : typ_env) (K : proc_env)
  (X : proc_var) (pf : fresh K x),
  closedP Q → disjoint (bv P) (bv Q) → disjoint (bvP P) (bvP Q) →
  types (G (K,, X =: T @ pf)) P U → types G K Q T → types G K (substP P Q X) U.
```

5.2 Subject Reduction Theorem

Subject reduction ensures that if a closed well-typed session fires a step then the redex is well-typed.

Theorem 1 (Subject Reduction). *Let* $\mathcal{M}$ *be a closed session. If* $\emptyset \Vdash \mathcal{M} : T$ *and* $\mathcal{M} \xrightarrow{\alpha} \mathcal{M}'$ *then there is* T' *s.t.* $T \xrightarrow{\alpha} T'$ *and* $\emptyset \Vdash \mathcal{M}' : T'$.

The theorem is a by-product of two core results.

Lemma 10 (Action Preservation). *If* $\Gamma \Vdash M : T$ *and* $M \xrightarrow{\alpha} M'$ *then there is* T' *s.t.* $T \xrightarrow{\alpha} T'$.

The mechanisation of Lemma 10 proceeds by induction on the session transition. The type `lts` deploys the LTS of sessions of Figure 2.

```
Lemma lts_typ_ltsM : ∀ (M1 M2 : session) (a : proc_action) (G : typ_env) (H : proc_env)
  (T : typ),
  lts M1 a M2 → types_session G H M1 T → ∃ U, typ_lts T a U.
```

Lemma 11 (Type Preservation). *Let* $\mathcal{M}$ *be a closed session and assume that there are types* T, T' *and action* α *s.t.:*

(i) $\emptyset \Vdash \mathcal{M} : T$;
(ii) $\mathcal{M} \xrightarrow{\alpha} \mathcal{M}'$;
(iii) $T \xrightarrow{\alpha} T'$.

We have $\emptyset \Vdash \mathcal{M}' : T'$.

The mechanisation of Lemma 11 proceeds by induction on the session transition, and relies on the lemmas introduced in the paper and on additional results dealing with the mechanism of shifting bound identifiers in the LTS of sessions (cf. § 2.3).

The predicate `closedS` holds when all processes in the session parameter are closed, while the predicate `wfm` holds when all processes in the session parameter do not reuse bound identifiers.

```
Lemma subject_reduction (M M' : session) (a : proc_action) (U U' : typ):
  closedS M → wfm M → types_session empty_env empty_penv M U → lts M a M' →
  typ_lts U a U' → types_session empty_env empty_penv M' U'.
```

6 Related and Future Work

To the best of our knowledge, only few works follow an iso-recursive approach to session types.

Our work In recent papers [25,26] we present iso-recursive multiparty sessions relying on a terminating function that decides the soundness of the environments inferred from the processes' types, and we mechanise both the function and the typing system in tools of the OCaml ecosystem relying on Why3 [42,11,15].

The approach presented in this paper complements the results of [25]:

This paper A type system for iso-recursive binary session types mechanised in Rocq. To type check a process composition, it is required to provide a proof that the dual types assigned to the processes are included in *type congruence*. This proof technique is reminiscent of that used for bisimilarity [37], while type congruence is more syntactical.

[25] A type system for iso-recursive multiparty session types mechanised in Why3. To type check a process composition, it is required to execute a terminating function that decides if the environment assigned to the process composition satisfies *compliance*. We showcase the implementation and verification of the function in OCaml by decorating the code with GOSPEL annotations [42,11].

Related work The paper [32] 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. [9,52]). The typing rules correspond to the proof rules in [4], 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 [54,55,56,57,47,41,35] 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 [7,3] 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 for modelling objects [1]. Pierce [44] further discusses the differences between iso-recursive and equi-recursive types.

Future work Concurrent actors and object programming are inherently *eager*: computations proceed by delegating and dividing a task to distributed actors which are running in parallel independently [2]. Our future work includes incorporating iso-recursive multiparty session types [25] into parameterised concurrent multiparty session types by Agha [10] to model and specify asynchronous multi-actor interactions.

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. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. The MIT Press (12 1986). https://doi.org/10.7551/mitpress/1086.001.0001
3. 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
4. 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
5. Barwell, A.D., Ferreira, F., Yoshida, N.: CONCUR test-of-time award for the period 1994-97 interview with Uwe Nestmann and Benjamin C. Pierce. J. Log. Algebraic Methods Program. **125**, 100744 (2022). https://doi.org/10.1016/J.JLAMP.2021.100744
6. 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
7. 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
8. 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
9. 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
10. Charalambides, M., Dinges, P., Agha, G.A.: Parameterized, concurrent session types for asynchronous multi-actor interactions. Sci. Comput. Program. **115-116**, 100–126 (2016). https://doi.org/10.1016/J.SCICO.2015.10.006
11. 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
12. Courcelle, B.: Fundamental properties of infinite trees. Theor. Comput. Sci. **25**, 95–169 (1983). https://doi.org/10.1016/0304-3975(83)90059-2

13. 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
14. 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
15. 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
16. 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
17. 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
18. 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
19. 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
20. Giunti, M.: GoPi: Compiling linear and static channels in Go. In: Bliudze, S., Bocchi, L. (eds.) Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12134, pp. 137–152. Springer (2020). https://doi.org/10.1007/978-3-030-50029-0_9
21. Giunti, M.: Iso-Recursive Session Types: a mechanisation in Rocq (2025), https://github.com/marcogiunti/Iso-Recursive-Session-Types
22. Giunti, M., Honda, K., Vasconcelos, V.T., Yoshida, N.: Session-based type discipline for pi calculus with matching. In: Programming Language Approaches to Concurrency and Communication (PLACES 2009) (2009), http://ctp.di.fct.unl.pt/~mgiunti/places09.pdf
23. 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
24. 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

25. Giunti, M., Yoshida, N.: Iso-recursive multiparty sessions and their automated verification. In: Vafeiadis, V. (ed.) Programming Languages and Systems - 34th European Symposium on Programming, ESOP 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part I. Lecture Notes in Computer Science, vol. 15694, pp. 349–378. Springer (2025). https://doi.org/10.1007/978-3-031-91118-7_14

26. Giunti, M., Yoshida, N.: Iso-Recursive Multiparty Sessions and their Automated Verification – Technical Report. CoRR **abs/2501.17778** (2025). https://doi.org/10.48550/ARXIV.2501.17778

27. 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

28. 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

29. Honda, K., Tokoro, M.: An object calculus for asynchronous communication. In: America, P. (ed.) ECOOP'91 European Conference on Object-Oriented Programming, Geneva, Switzerland, July 15-19, 1991, Proceedings. Lecture Notes in Computer Science, vol. 512, pp. 133–147. Springer (1991). https://doi.org/10.1007/BFB0057019

30. 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

31. Honda, K., Yoshida, N.: On reduction-based process semantics. Theor. Comput. Sci. **151**(2), 437–486 (1995). https://doi.org/10.1016/0304-3975(95)00074-7

32. 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

33. 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

34. 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

35. 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
36. McBride, C.: Elimination with a motive. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers. Lecture Notes in Computer Science, vol. 2277, pp. 197–216. Springer (2000). https://doi.org/10.1007/3-540-45842-5_13
37. Milner, R.: Communication and concurrency. PHI Series in computer science, Prentice Hall (1989)
38. Milner, R.: Functions as processes. Math. Struct. Comput. Sci. **2**(2), 119–141 (1992). https://doi.org/10.1017/S0960129500001407
39. Milner, R.: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press (1999)
40. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, Parts I and II. Info.& Comp. **100**(1) (1992)
41. 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
42. 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
43. 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
44. Pierce, B.C.: Types and programming languages. MIT Press (2002)
45. Pous, D.: Up-to techniques for weak bisimulation. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3580, pp. 730–741. Springer (2005). https://doi.org/10.1007/11523468_59, https://github.com/rocq-archive/weak-up-to
46. Rocq Team: Reference manual, https://coq.inria.fr/doc/V8.20.0/refman/
47. Rossberg, A.: Mutually iso-recursive subtyping. Proc. ACM Program. Lang. **7**(OOPSLA2), 347–373 (2023). https://doi.org/10.1145/3622809
48. 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
49. 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
50. 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. Lec-

ture Notes in Computer Science, vol. 817, pp. 398–413. Springer (1994). https://doi.org/10.1007/3-540-58184-7_118
51. Vasconcelos, V.T., Giunti, M., Yoshida, N., Honda, K.: Type safety without subject reduction for session types (2010), http://ctp.di.fct.unl.pt/~mgiunti/vgyh_ts_without_sr.pdf
52. Wadler, P.: Propositions as sessions. J. Funct. Program. **24**(2-3), 384–418 (2014). https://doi.org/10.1017/S095679681400001X
53. Yonezawa, A., Tokoro, M.: Object-oriented concurrent programming. MIT Press, Cambridge, MA, USA (1986)
54. 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
55. 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
56. 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
57. 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